Available online at www.sciencedirect.com

ScienceDirect

Computer Communications 31 (2008) 668–698

computer communications

www.elsevier.com/locate/comcom

Efficiently reconfigurable backbones for wireless sensor networks ☆

Stefano Basagni [a,*], Chiara Petrioli [b], Roberto Petroccia [b]

[a] *ECE Department, 312 Dana, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA*
[b] *Università di Roma La Sapienza, Dipartimento di Informatica, via Salaria 113, 00198 Roma, Italy*

Available online 24 October 2007

Abstract

We present the definition and performance evaluation of a protocol for building and maintaining a connected backbone among the nodes of a wireless sensor networks (WSN). Building backbones first, and then coping with network dynamics is typical of protocols for backbone formation. Rules for building the backbone, however, do not take into account the following network dynamics explicitly. This makes maintaining a connected backbone quite costly, especially in terms of reorganization time, overhead and energy consumption. Our protocol includes in the backbone forming operations a fail-safe mechanism for dealing with the addition and the removal of nodes, which are typical events in a WSN. More specifically, the network is kept partitioned into clusters that are *cliques*, i.e., nodes in each cluster are directly connected to each others. Therefore, removing a node does not disrupt a cluster, and adding one requires simple operations for checking node admission to the cluster. The protocol, termed CC ("double C", for *clique clustering*), comprises three phases, each designed to render the operations of the others swift and efficient. The first phase partitions the network into clusters that are cliques. Clusters are then joined to form a backbone that is provably connected. Finally, the third, more on-line phase, maintains the backbone connected in face of node additions and removals. We compare the performance of CC with that of DMAC, a protocol that has been previously proposed for building and maintaining clusters and backbones in presence of network dynamics. Our comparison concerns metrics that are central to WSN research, such as time for clustering and backbone reorganization, corresponding overhead, extent of the reorganization (i.e., number of nodes involved in it), and properties of the resulting backbone, such as its size, backbone route length, number of gateways and nodes per cluster. Our ns2-based simulation results show that the design criteria chosen for CC are effective in producing backbones that can be reconfigured quickly and with remarkably lower overhead.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Wireless sensor networks; Clustering; Backbone formation

1. Introduction

The extensive research and experimentation performed in the past couple of decades witness the remarkable interest of the academic and industrial community in wireless ad hoc [1] and sensor networks (WSNs) [2,3], and in protocols that make possible to deploy these networks effectively and at reasonable cost. Among the topics that have received particular attention, *clustering formation and interconnection*, also referred as *backbone formation*, has triggered quite a community-wide discussion. The reasons are to be found in the traditional use of a superimposed hierarchical structure over the "flat" network topology for favoring routing scalability in terms of routing table size, reduced routing overhead, and for other benefits. There are also reasons that pertain specifically to wireless multi-hop networks such as ad hoc and WSNs. In the latter case, especially, clustering provides a natural choice of selecting aggregation points for performing data fusion. It has been demonstrated that the beneficial effects of data fusion largely counteracts the sub-optimality of routes over the backbone [4]. Network partitioning provides also a straightforward way for topology control. Once clusters and a backbone have been formed, a few nodes per each

☆ This material is based upon work partially supported by the MIUR International FIRB RBIN047MH9.
* Corresponding author. Tel.: +1 617 373 3061; fax: +1 617 373 8970.
E-mail addresses: basagni@ece.neu.edu (S. Basagni), petrioli@di.uniroma1.it (C. Petrioli), petroccia@di.uniroma1.it (R. Petroccia).

cluster (the *clusterhead* and the *gateway nodes* that provide inter-cluster connection) remain awake to perform the network operations, while the radio interface of all the other nodes is turned off (sleep mode) for energy saving and prolonged network lifetime [5].

The majority of backbone formation protocols proposed for WSNs have been designed for static, or quasi static networks. This implies that when nodes move away, or fail, the backbone formation process must be re-executed. Recently some protocols have been proposed for ad hoc networks [6] and for WSNs [5] that explicitly cope with these different types of network dynamics without repeating the whole clustering formation and interconnection protocol. Additional procedures are usually provided that react to the presence of new links (like those formed by nodes moving closer) and to link failure (due to nodal mobility, node failure, etc.). Clustering maintenance, as well as interconnection maintenance, however impose non-negligible overhead, and decrease the overall network performance.

All these backbone formation schemes (and the large majority of the ones proposed so far) form clusters and backbones where the nodes are at most one hop from their clusterhead. This is effective in reducing the maintenance overhead when a node fails or moves away from the network (for quite an extensive list of works on clustering, see [7]). However, while the set of clusterheads dominates all other nodes, no guarantee is provided about the possibility of other nodes in a cluster to be able to directly communicate. This implies longer routes. It also increases overhead in case, say, a clusterhead is removed (e.g., it dies because of energy depletion or other failures). In this case a new clusterhead has to be selected and the cluster nodes have to decide whether to affiliate to the new clusterhead, to join an adjacent cluster or to become clusterheads themselves. Changes in the number and structure of clusters may also result in the need to select new gateway nodes to join adjacent clusters and maintain the backbone connected. If the clusterhead is selected based on a criterion which favors the best node to serve in this role (e.g., the one with the highest residual energy), changes in a node affiliation may result in the need to change the current clusterhead and gateways. This might in turn trigger a "chain reaction" of changes in response to a single failure, which imposes high maintenance overhead.

In this paper we introduce and evaluate a new protocol where coping with nodal failures and with the addition of new nodes is embedded in the design of the backbone formation protocol. The idea is that of *building and connecting* clusters that are resilient to the failures of one or more nodes. The new protocol, termed *clique clustering*, or CC (*double C*) for short, is comprised of three phases that are integrated seamlessly to provide network partition, cluster interconnection and backbone reorganization. The first two phases are also referred as backbone formation phase. The protocol is executed at each node (i.e., it is distributed) based on simple assumptions and on information about the node immediate neighbors (i.e., it is localized). The first phase of CC, the *cluster formation* phase, partitions the network into clusters that are *cliques*, i.e., subsets of the network nodes that are all connected to each other. Once the clusterheads have been selected a backbone is efficiently formed that is connected (second phase: *cluster interconnection*). The third, more on-line, phase of CC takes care of node addition and removal. This phase is implemented by a few compact procedures that provide CC with an efficient and low-overhead mechanism for dealing with the case when new, fresh nodes are added to the network or nodes are removed because of failure.

We demonstrate CC through simulations. In particular, we show its effectiveness in quickly and efficiently responding to nodal removal and addition by comparing it with DMAC, a clustering protocol for which backbone construction and reorganization have been defined and tested. The Distributed Mobility-Adaptive Clustering (DMAC) [6] was originally proposed to extend clustering protocols for static multi-hop scenarios with mechanisms for coping with the mobility of nodes. The investigation of several backbone formation solutions for large multi-hop networks such as the one we pursued in [7], shows that DMAC is quite a promising protocol for WSNs characterized by node addition and removal. However, for the way it was defined, DMAC also gives an example of how the removal or addition of a node can trigger a chain reaction involving nodes in the network that can be quite far from the one removed/added. As such, the overhead associated to backbone maintenance imposes quite a severe toll on network functionalities. CC aims at avoiding these pitfalls by dealing with nodal removals and additions right where they happen.

CC and DMAC are compared with respect to metrics that show how building clusters that are cliques is effective in reducing time, overhead and the number of nodes involved in a reorganization while producing reasonably small backbones. Metrics of interest to our comparison include the following: time it takes to re-build a connected backbone, overhead for cluster construction and maintenance, number of nodes involved in the backbone reorganization, backbone size and route length. We observed that while producing backbones bigger than DMAC, CC reacts to network changes efficiently and quickly. More precisely, DMAC incurs remarkably higher overhead than CC, and needs way longer time to reconfigure after nodal removal and addition. Therefore, CC provides a suitable solution for organizing dynamic WSNs hierarchically in realistic scenarios, where overhead-imposed energy consumption and reorganization time must be minimized.

The paper is organized as follows: previous works on partitioning a multi-hop network into cliques are reviewed in Section 2. Section 3 describes our CC protocol and its three phases in detail. We give a brief overview of DMAC in Section 4. The performance comparison between CC and DMAC is investigated in Section 5. Finally, Section 6 concludes the paper.

2. Related work

The problem of grouping the nodes of a multi-hop network in clusters, interconnected via gateways into connected backbones for increased scalability and overall protocol efficiency, has been widely investigated both in the context of ad hoc networking research, and, more recently, for WSNs. A quite exhaustive survey on these topics can be found in [7], where a performance-based comparison of leading solutions for WSN backbone formation is also presented. Many recent papers, such as [8–10], witness the strong interest of the research community in this topic. The majority of the clustering schemes proposed for WSNs (including the ones evaluated in [7]) works for static networks. In other words, they cannot deal with network dynamics such as nodes mobility, add or removal of nodes, failures and "death" of nodes due to energy depletion. For most of the WSN applications dealing with mobility is still largely unnecessary. However, the resource-constrained nature of wireless sensor nodes imposes to design protocols that are able to efficiently and effectively deal with node removal, mostly because of death or failure, and additions, when new batches of fresh nodes are added to replace depleted ones and prolong network functionalities. Therefore, protocols are needed that are able to efficiently set up and maintain a clustering structure and a corresponding connected backbone while effectively reacting to unavoidable WSN dynamics. The topic we deal with in this paper concerns showing that a clique-based backbone formation and maintenance can provide an effective solution for dealing with these dynamics. Furthermore, we demonstrate that this approach outperforms other well-known and established connected dominating sets-based approaches to clustering, which also deal with backbone reorganization after nodal removal/addition, such as DMAC [6]. The reasons we looked at DMAC stem from our previous performance evaluation-based investigation of backbone formation solutions for WSNs [7]. There we showed that the clustering formation and interconnection parts of the DMAC protocol, i.e., its core, provide an excellent starting point to partition the network efficiently, and incurring in low formation times and overhead.

Partitioning a multi-hop network into cliques has been investigated before, and papers can be found where distributed solutions are given with different properties. One of the first works on the subject is [11], where clustering is used for making routing in dynamic (ad hoc) networks more scalable. More precisely, the authors investigate the partitioning of the network into clusters of diameter k, with the particular case of a cluster being a clique when $k = 1$. The focus of this paper is demonstrating the effectiveness of the presented ad hoc routing protocols in delivering packets among pairs of selected nodes, and the clustering is seen as a mean of favoring scalability. In other words, clustering is here only functional to routing and the description of the clustering organization, the analysis of its properties and its demonstration is not presented. Furthermore, in dense scenarios, this solution is not practical, since it calculates all cliques a node can belong to and this computation generates quite a remarkable amount of overhead, with a detrimental effect on network performance.

An approach similar to that of [11] is explored in [12], where clustering starts with the construction of a spanning tree over the network nodes. Clusters are then built in a distributed way so that two nodes in a cluster are at most k hops apart. When $k = 1$ this protocol clearly generates cliques. The motivations of this work concern enabling scalable routing. Differently from [11], clustering properties are investigated. It is shown that the proposed protocol approximates the lowest number k-clustering (i.e., a clustering with the lowest number of clusters of diameter at most k) with a competitive ratio of $O(k)$. The proposed algorithm concerns setting up clusters on a static network with no nodal removals or additions.

A distributed algorithm for partitioning a *multi-agent system* (MAS) where agents and their interconnections are represented by a graph is described in [13] (ad hoc networks, and hence WSNs, are particular MASs). Every node v computes all the cliques it belongs to. Then v selects the biggest clique and checks whether all the nodes in that clique agree to affiliate to it. If this is the case, a cluster is formed, and its nodes communicate this information to their neighbors. Upon receiving this information, every other node recomputes its cliques based on the updated information. If the biggest clique of a node cannot be formed, the node updates the information it has and selects the next biggest clique, repeating the process until it finds the clique it can participate in. The efficiency of this protocol depends on how large a clique can be. For this reasons, the authors limit the size of the cliques to contain the corresponding protocol duration and overhead. Computing all cliques (up to a given size) and repeatedly advertising the current biggest clique generate non-negligible overhead. Furthermore, the presented protocol is not suitable for dynamic networks. Node coalitions (i.e., cliques) in MASs are simply disrupted when a node moves away or fails. A clustering method similar to the one presented in [13] is also used in [14] and [15]. Determining the cliques a node belongs to is aided by the knowledge of geographic information (each node knows its coordinates) in [14]. In this case each node finds its cliques by computing the cliques around every edge, and then it decides to participate to the biggest one, if possible. Although distributed, and hence suitable for WSNs, this algorithm makes some strong assumptions. For instance, it is assumed that all the nodes around a link are easily found. The protocol has no provisions for coping with any kind of network dynamics. In [15] an extra step is added to the basic steps of the algorithm of [13] for securing the clustering construction. Upon determining its membership to a clique each node performs a conformity check for identifying potential malicious nodes. If no such node is found, nodes enforce a clique agreement and exit the protocol. Otherwise, the malicious node is removed from the network and the nodes start the protocol from scratch. As for similar solutions, this protocol has no methods for dealing with mobility or node failure.

The problem of efficiently computing all maximal cliques in a unit disk graph (UDG, a common model for static wireless ad hoc and sensor networks [16]) is investigated in [17]. In this work the focus is that of finding (a centralized) polynomial time approximation algorithm for producing all maximal cliques by using key geometric structures of UDGs, which in turn requires each node to be aware of their location. Specifically, for each edge in the graph, the set of nodes that are included in the area delimited by some given shapes built starting from that edge are considered. It is then shown that all possible cliques having this as the longest edge are contained in that area. An algorithm is presented to select some of these cliques as the clusters, and the authors also indicate that the proposed algorithm can be distributed. No indication on how the algorithm reacts to network dynamics is provided.

All the above algorithms about clustering by cliques are concerned with building the clusters per se, or with using the resulting network partitioning to favor routing, security, etc. None of these solutions has been extended to include explicit backbone formation, or to incorporate features for automatically counteracting the effect of possible failures (mobility, battery depletion, etc.) and addition. This prompted us to come up with a protocol which builds clique-clusters and connect them into a backbone which is fail-safe to network dynamics such as node removal/addition while incurring reasonable overhead.

3. Clique clustering (CC)

3.1. Cluster formation phase

The cluster formation phase of the CC protocol produces a clustering that satisfies the following properties: (1) every non-clusterhead node (often referred to as *ordinary node*) has at least a clusterhead as neighbor (i.e., the set of clusterheads is a dominating set); (2) every node in a cluster can communicate directly with every other node in the cluster (clique property), and (3) every non-clusterhead node affiliates to the cluster of the first clusterhead inviting it.

In describing CC we assume that every node knows its own unique identifier (ID), its own *weight* and the ID and weight of each of its neighbors. The weight of a node is a real number $\geqslant 0$ which depends on the node current status and application requirements, and that indicates the suitability of the node for being selected as a clusterhead. The higher the weight the better is a node for assuming that role. The protocol is executed at each node v in such a way that v decides its role (clusterhead or ordinary node) as soon as its "heavier" neighbors (neighbors with bigger weight) have decided their own role. The protocol is started by those nodes that have the biggest weight among all their neighbors (called *init nodes*). These nodes send a (broadcast) message telling they will be clusterheads. Upon receiving this message from one of its heavier neighbors, a node exchanges with the sender information about its own neighbors. Based on the received information, a clusterhead selects all those smaller neighbors that can be affiliated to its own cluster while maintaining the clique property, and invites them to join it. A node whose heavier neighbors have joined other clusters or have finished inviting nodes, and that has not been invited to be part of any cluster decides to be a clusterhead itself. The protocol terminates when every node belongs to a cluster being either a clusterhead or an ordinary node and knows the role and clusterhead of all its neighbors.

Except for the initial procedure, which is executed by each node when it starts the protocol operations, the cluster formation phase of CC is message-triggered. We have six types of messages: (1) CH(v). Sent by a node v to declare that it will be a clusterhead. This is a broadcast message, i.e., a message that has to reach all v's neighbors. (2) NEIGHBORS(v,u). Sent by a node v to its neighbor u to communicate v's neighbor list. This is a unicast message from node v to node u. (3) ASSOC_REQUEST(v,u). Node v invites node u to be part of its cluster by sending to node u this unicast message. Nodes are invited according to their degree: Nodes with higher degree are invited first. (4) ASSOC_RESPONSE(u,z). Sent by a node v to its neighbor u to communicate v's response to u's association request. If the response is **yes** node v communicates to its clusterhead u to be part of u's cluster ($z = u$), otherwise node v communicates to a clusterhead u to be part of z's cluster ($z \neq u$). (5) ASSOC_FINISH(v). Sent by a node v to declare that it has concluded its cluster formation. This is a broadcast message. With this message, a clusterhead declares that it has completed its cluster. It will not try to enroll any other nodes. (6) JOIN(v,u). Sent by a node v to declare that it is now associated to clusterhead u. This is a broadcast message.

In the description of the procedures we will use the following notation:

- v is the ID of a generic node executing the protocol.
- w_v is the weight of node v.
- For each node v, *Neigh*(v) indicates the set of node v's neighbors, i.e., all the nodes that can receive messages from v.
- *d*(v) is node v's degree, i.e., the number of its neighbors. Clearly, $d(v) = |Neigh(v)|$.
- *Clusterhead*(v) contains the ID of the clusterhead to which node v is affiliated. If node v is a clusterhead then *Clusterhead*(v) = v. It is initialized to NULL.
- *Cluster*(v) lists the nodes in the cluster of node v. All nodes in the same cluster have the same *Cluster* set. It is initialized to $\emptyset$.
- *Flag*(u), where $u \in Neigh(v)$, is set to **true** only when node v is made aware that node u has decided its role and cluster. It is initialized to **false**.
- *Ch*(v) is set to **true** only if node v is a clusterhead. It is initialized to **false**.

- *update_Info*(u) is executed by a node v to record ID and weight of node u.
- *store_neigh*(u) is executed by a clusterhead v to record u's neighbor list after receiving NEIGHBORS(u, v).
- *getNextNodeToAssociate*() is executed by a clusterhead v to select the next node to associate to its cluster. If no other node can be associate, it returns -1. This function returns -1 when either all neighbors are in a cluster or no other nodes can be invited.
- **EXIT** is executed by a node for terminating the execution of the first phase of CC. This happens when the node has decided its role and knows all the needed information about its neighbors.

Every node starts the protocol by executing the procedure *InitCC* below. Only nodes that have the highest weight among all their neighbors (called *init nodes*) will send a CH message. Given the nature of nodal weights (real numbers $\geqslant 0$) it always exists at least a node v that transmits CH(v). All other nodes just wait to receive a message.

Algorithm 1. InitCC {Executed by every node v}

```
if (∀ u ∈ Neigh(v): $w_v > w_u$) then
    send CH(v);
    Cluster(v) :=Cluster(v) ∪ {v};
    Ch(v) :=true;
    Clusterhead(v) :=v;
if (Neigh(v) = ∅) then
    send ASSOC_FINISH(v);
    EXIT;
```

The operations executed by an init node are: Sending the broadcast message CH(v), updating its variable *Cluster* from $\emptyset$ to $\{v\}$, setting *Ch*(v) to **true** and setting *Clusterhead* of node v to v. In case node v is the only node in the network, or in case v is an isolated node (in both cases v has no neighbors), it simply communicates that it is done and exits the execution of this phase of CC.

Upon receiving the message CH(u) from a neighbor u, node v executes the procedure On receiving CH(u) (Algorithm 2). Node v records the information about u and, if u is a heavier neighbor, it unicasts to node u the list of its own neighbors. Node u needs to receive this list for deciding whether to invite v in its cluster or not. If node v is already affiliated with a clusterhead, it sends a "dummy" (empty) NEIGHBORS message, so that node u does not invite v.

Algorithm 2. On receiving CH(u) {Node v receives CH from u}

```
update_Info(u);
Ch(u) :=true;
Clusterhead(u) :=u;
if ($w_v < w_u$) then
    send NEIGHBORS(v,u);
```

Procedure On receiving NEIGHBORS(u, v) (Algorithm 3) is executed by a node v that has previously sent a message CH(v) when it receives from a neighbor u the message NEIGHBORS(u, v).

Algorithm 3. On receiving NEIGHBORS(u, v) {Node v receives NEIGHBORS from u}

```
store_neigh(u);
if (received neighbor list from all smaller neighbors)
then
    x: = getNextNodeToAssociate();
    if ($x \neq -1$) then
        send ASSOC_REQUEST(v,x);
    else
        send ASSOC_FINISH(v);
        if (∀z ∈ Neigh(v):Flag(z)= true) then
            EXIT;
```

Node v records u's neighbor list (*store_neigh*(u)). Then it checks whether all its neighbors z have sent their neighbor list. In the affirmative, node v checks which of its smaller neighbors can be invited to be part of its cluster. This is accomplished by executing procedure *getNextNodeToAssociate*() (described below) with which node v selects the next node to invite to its cluster among all nodes that belong to the intersection of the neighbor lists of the nodes that already affiliated with its cluster (the cluster grows, while maintaining the clique property). These nodes should not also belong to a cluster already (*Flag* = **false**). Within the set of the nodes with these properties, node v selects a node with the maximum degree (this rule aims at trying to create clusters with the highest number of nodes). Ties are broken based on nodal weights and IDs.

Algorithm 4. getNextNodeToAssociate() {Clusterhead v selects the next node to invite to its cluster}

```
if ({u:u ∈ Neigh(v) and $w_u < w_v$ and
  u ∈ $\bigcap_{z \in Cluster(v)}$Neigh(z) and Flag(u) = false} ≠ ∅)
  then
  x: = $\max_{d(\cdot)}${u:u ∈ Neigh(v) and $w_u < w_v$ and
u ∈ $\bigcap_{z \in Cluster(v)}$Neigh(z) and Flag(u) = false};
  return x;
else
return −1;
```

Node x is invited to be part of node v's cluster by sending the (unicast) message ASSOC_REQUEST(v, x). If no node x exists that belongs to the neighbor lists intersection, node v cannot invite anymore nodes without violating the clique property. At this time, it sends the message ASSOC_FINISH(v) to communicate to its neighbors that it will not enroll anymore nodes. If node v knows the information about the role and the cluster of all its neighbors it terminates the execution of CC executing **EXIT**.

When a node v receives an invitation to join node u's cluster, i.e., v receives the message ASSOC_REQUES-

T(u, v), it executes the procedure On receiving ASSOC_REQUEST(u, v) described below (Algorithm 5). If node v is not associated to any cluster ($Clusterhead(v) = \text{NULL}$), it accepts u's request by transmitting ASSOC_RESPONSE(u, u). It then communicates its membership to u's cluster to all its neighbors, broadcasting JOIN(v, u) with information about the ID and weight of u and v. If node v is already affiliated to a cluster, it sends ASSOC_RESPONSE(u, $Clusterhead(v)$) to u. (Notice that in this phase of CC a node always joins the cluster of the first clusterhead that invites it.)

Algorithm 5. On receiving ASSOC_REQUEST(u, v) {Node v receives ASSOC_REQUEST from u}

```
if ($Clusterhead(v) = \text{NULL}$) then
    $Clusterhead(v) := u$;
    send ASSOC_RESPONSE($u, u$);
    send JOIN($v, u$);
else
    send ASSOC_RESPONSE($u$, $Clusterhead(v)$);
```

When a clusterhead v receives the message ASSOC_RESPONSE(u,z) (Algorithm 6), i.e., it receives a response to an association request from node u, it checks whether $v = z$. In the affirmative, node v adds u to its cluster and keeps inviting nodes. In the negative, v keeps inviting other nodes, if any. If no new node can be invited to v's cluster, node v sends the message ASSOC_FINISH(v) to communicate this information to its neighbors. If v knows the information about the role and the cluster of all its neighbors it terminates the execution of the first phase of CC executing **EXIT**.

When a node v receives a message ASSOC_FINISH(u) from a neighboring clusterhead u, it sets $Flag(u)$ to **true** and checks whether it is already part of a cluster or not (Algorithm 7). When this is the case, if it knows the information about the role and the cluster of all its neighbors it terminates the execution of this phase of CC executing **EXIT**. In case v still does not belong to any cluster it checks if all its heavier neighbors t have decided their role. In the positive it decides to be a clusterhead and communicates its decision to all its neighbors, updating also the needed variables/data structures. In case node v does not have any smaller neighbors, it does not have to wait, and simply sends an ASSOC_FINISH.

Upon receiving a message JOIN(u, z) from a neighbor u that is associated to z's cluster, node v executes the procedure On receiving JOIN(u,z). Node v sets $Flag(u)$ to **true**, since now it is aware of the role and the cluster of u. It also records information about u's clusterhead, $Clusterhead(u) := z$, and keeps track of the ID and the weight of u and z. If v is not affiliated to a cluster ($Clusterhead(v) = \text{NULL}$) it checks if all neighbors with higher weight have a role and a cluster. In the affirmative, node v becomes a clusterhead, broadcast this information to its neighbors and then, after the reception of all NEIGHBORS messages, starts creating its cluster. In case node v does not have any smaller neighbors, it can simply send an ASSOC_FINISH. In the negative, v checks if all its neighbors have a role and a cluster to exit the execution of the first phase of CC.

Algorithm 6. On receiving ASSOC_RESPONSE(u, z) {Node v receives ASSOC_RESPONSE from u where z is the clusterhead of u}

```
if ($v = z$) then
    $Cluster(v) := Cluster(v) \cup \{u\}$;
$Clusterhead(u) := z$;
$x := getNextNodeToAssociate()$;
if ($x \neq -1$)
    send ASSOC_REQUEST(x);
else
    send ASSOC_FINISH($v$);
    if ($\forall t \in Neigh(v)$:$Flag(t)$= true) then
        EXIT;
```

Algorithm 7. On receiving ASSOC_FINISH(u) {Node v receives ASSOC_FINISH from u}

```
$Flag(u)$ :=true;
if ($Clusterhead(v) = \text{NULL}$) then
    if ($\{t{:}t \in Neigh(v)$ and $w_t > w_v$ and $Flag(t) =$
            false$\} = \emptyset$) then
        send CH(v);
        $Cluster(v) := Cluster(v) \cup \{v\}$;
        $Ch(v)$ :=true;
        $Clusterhead(v) := v$;
        if ($\{t{:}t \in Neigh(v)$ and $w_v > w_t\} = \emptyset$) then
            send ASSOC_FINISH($v$);
            EXIT;
else
    if ($\forall z \in Neigh(v)$:$Flag(z)$= true) then
        if ($Clusterhead(v) \neq v$) then
            $Cluster(v) := \{x{:}x \in Neigh(v)$ and
                $Clusterhead(x) = Clusterhead(v)\} \cup \{v\}$;
        EXIT;
```

Algorithm 8. On receiving JOIN(u,z) {Node v receives information about node u joining z}

```
$Flag(u)$ :=true;
$Clusterhead(u) := z$;
$update\_Info(u)$;
$update\_Info(z)$;
if ($Clusterhead(v) = \text{NULL}$) then
    if ($\{t{:}t \in Neigh(v)$ and $w_t > w_v$ and $Flag(t) =$
            false$\} = \emptyset$) then
        send CH(v);
        $Cluster(v) := Cluster(v) \cup \{v\}$;
        $Ch(v)$ :=true;
        $Clusterhead(v) := v$;
        if($\{t{:}t \in Neigh(v)$ and $w_v > w_t\} = \emptyset$) then
        send ASSOC_FINISH($v$);
        EXIT;
```

```
else
  if ($\forall t \in Neigh(v)$:$Flag(t)$= true) then
    if ($Clusterhead(v) \neq v$) then
      $Cluster(v) := \{x : x \in Neigh(v)$ and
        $Clusterhead(x) = Clusterhead(v)\} \cup \{v\}$;
    EXIT;
```

We observe that when a node v exits the execution of the cluster formation phase it knows the weight of all its neighbors, the ID and the weight of their clusterheads, and hence the cluster to which its neighbors are associated. Each node knows also the list of all nodes in its own cluster.

Example. Let us consider the ad hoc network depicted in Fig. 1(a), where the numbers associated to the nodes indicate both nodes ID and weight. All nodes execute procedure *InitCC* to determine whether they are the bigger nodes in their neighborhood. Being the heaviest nodes in their neighborhood, only nodes 11, 15 and 18 are init nodes, and hence they send a CH message. In particular, nodes 8, 9 and 10 receive the message CH(11) from their neighbor 11, nodes 16 and 17 receive CH(18) from 18 and nodes 12, 13 and 14 receive CH(15). All nodes 8, 9, 10, 12, 13, 14, 16 and 17 reply to the CH message by sending the list of their neighbors (with a message NEIGHBORS). Upon receiving NEIGHBORS(8, 11), NEIGHBORS(9, 11) and NEIGHBORS(10, 11), node 11 starts to select nodes to affiliate them to its cluster. Node 8 is the neighbor with the higher degree; therefore, node 11 invites node 8 by unicasting the message ASSOC_REQUEST(11, 8). Since node 8 has no clusterhead yet, it decides to affiliate to node 11, sends ASSOC_RESPONSE(8, 11) to 11 and broadcasts the message JOIN(8, 11). Node 11, after reception of ASSOC_RESPONSE(8, 11) sees that 8 accepted its request and moves to invite other nodes. Upon receiving the message JOIN, neighbors of node 8 update information about node 8 and 11. Node 11, then invites first node 10 and then node 9, since inviting both of them would not violate the clique property. Node 10, like 8, accepts, by unicasting ASSOC_RESPONSE(10, 11), and sends the message JOIN(10,11) to all its neighbors. Later, node 9 accepts 11's invitation and informs all neighbors about enrolling in 11's cluster. At this point, 11 cannot invite other nodes and sends message ASSOC_FINISH(11) to all its neighbors. Since 11 now knows the role and the cluster of all its neighbors it can **EXIT** the execution of CC. Nodes 9 and 10 can also **EXIT** the protocol. Before exiting, node 8 has to wait for the decision of its neighbors. Similarly, nodes 15 and 18 form their clusters, containing nodes 12, 13 and 14 and 16 and 17, respectively. In the meanwhile, node 7 has been made aware of the role and the clusterhead of its heavier neighbor 8 (via a JOIN(8,11) message) and knows that node 8 will not be a clusterhead. Node 7 has no choice but becoming a clusterhead itself and hence it broadcasts the message CH(7) to which those neighbors that have not decided what to do (nodes 4, 5 and 6) reply

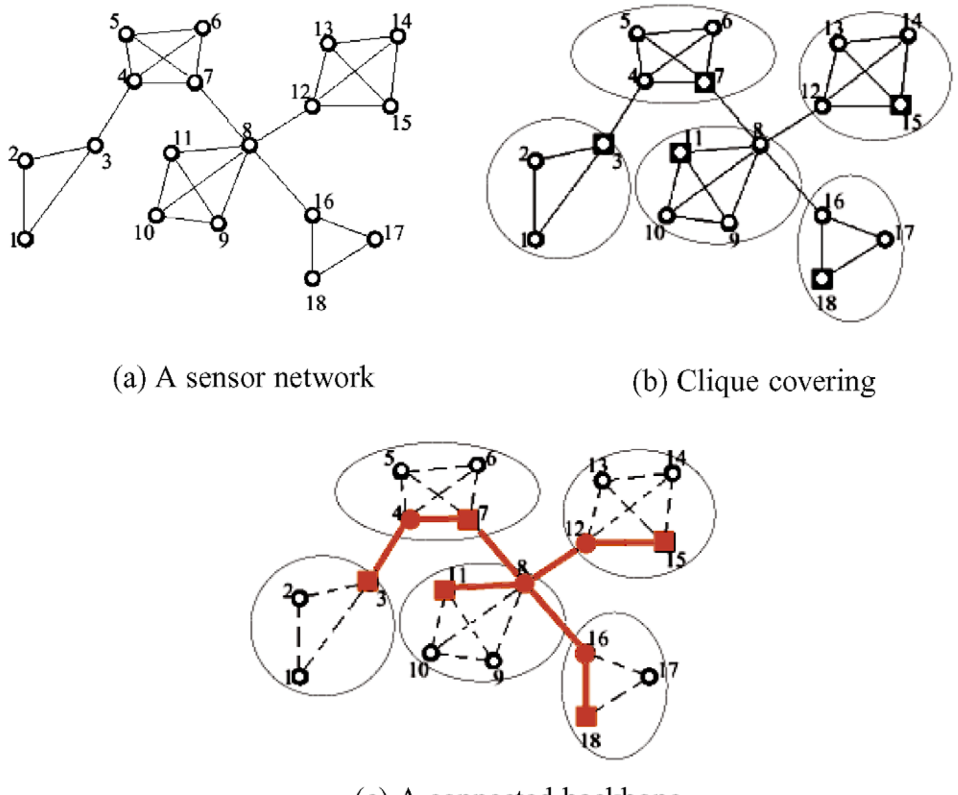


Fig. 1. A WSN, the CC-induced clustering and a backbone connecting the clusters.

by sending their neighbor lists. Based on these lists, node 7 will invite, in the order, nodes 4, 6 and 5 and then, after having updated its cluster, quits. By accepting node 7's invitation and by sending message JOIN(4, 7), node 4 "frees" node 3 from waiting any longer. The last cluster is formed with node 3 as clusterhead and nodes 1 and 2 as its affiliated nodes. The result of the execution of the first phase of CC is shown in Fig. 1(b).

3.1.1. Cluster formation correctness

We prove that the distributed execution of the clique formation phase partitions the network into clusters that are cliques within a finite time. We reasonably assume that pairwise message exchange is always successful, and that it happens in a bounded amount of time. As a first step, we show that a node assumes one of the two roles of clusterhead or ordinary node in finite time (Proposition 1). A node becomes formally a clusterhead when it has completed the formation of its cluster, namely, after it has decided to be a clusterhead, it has sent the corresponding CH message, it has invited some of its neighbors to form a clique and it has received positive/negative replies from them, being therefore able to complete its cluster. A node becomes an ordinary node after it has been invited and it has decided to join a cluster.

We start by proving the following useful result.

Lemma 1. *During the first phase of CC whenever a node assumes a role (either clusterhead or ordinary node) it does not change it anymore.*

Proof. A node v decides to become a clusterhead by executing those parts of Algorithm 1, 7 or 8 where a CH message is sent. In every case the variable *Clusterhead*(v) (initialized to NULL) is set to v. A node v decides to become an ordinary node as soon as it receives the first ASSOC_REQUEST from a node $u \neq v$ and executes Algorithm 5. In this case, too, the variable *Clusterhead*(v) is set from NULL to u. By inspecting the code it is easy to verify that in order for a node v to change its role once it has gotten one, i.e., to set the variable *Clusterhead*(v) to something different from what it has been set to, either a node has to execute Algorithm 1 again or the variable *Clusterhead*(v) itself should be reset to NULL. Neither of the two cases can actually occur, since InitCC is executed once and for all at the beginning of the protocol operations, and all assignments to *Clusterhead*(v) are "(*Clusterhead*(v) = NULL)-protected." □

Proposition 1. *Every node assumes either the role of clusterhead or of ordinary node within a finite time.*

Proof. We proceed by induction on the following ordering. We start by listing the init nodes according to decreasing weights. We then list the remaining nodes according to their decreasing weights. (Ties, as usual, are broken based on the nodal unique ID.) We notice that this is a well defined ordering because of the total ordering among weights. The base case of the induction is made up of the init nodes. Clearly, by executing the InitCC procedure, an init node v sets its *Clusterhead*(v) variable to v and broadcasts the CH message. If not v does not have a smaller neighbor it immediately broadcasts an ASSOC_FINISH message. Otherwise, since messages do arrive within a finite time, each neighbor of v receives the CH and cannot do anything but responding with a NEIGHBOR message (Algorithm 2 is just a plain list of statements). These messages are received by v within a finite time, so that v can start sending ASSOC_REQUEST messages, and building its cluster (Algorithm 3). We observe that eventually, after a finite time, a clusterhead is bound to send an ASSOC_FINISH message (Algorithm 3).

Let us now assume that all nodes up to h in the ordering, $h >$ cardinality of the set of the init nodes, have assumed a role within a finite time (inductive hypothesis). This means that all nodes up to the hth have sent either an ASSOC_FINISH message (i.e, they are clusterheads) or a JOIN message (for ordinary nodes). Let us now consider node v, the $(h+1)$th node in the ordering. All v's bigger neighbors precede v in the ordering, and therefore, by induction, they have decided their role in finite time. Two are the cases now. All v's bigger neighbors have either joined some other cluster, and hence have sent a JOIN message, or they are clusterheads that have not invited v (and they sent an ASSOC_FINISH). In this case, node v decides to be a clusterhead itself, and sends a CH (Algorithms 7 and 8, first inner if). As for the case of an init node (base case), node v forms its cluster and sends an ASSOC_FINISH message within finite time. If instead node v has received an ASSOC_REQUEST, after sending an ASSOC_RESPONSE to the inviting clusterhead it sends a JOIN message. Therefore, in finite time it decides to be an ordinary node. Thanks to Lemma 1, once a node has assumed a role in finite time, it does not change it. □

We are now able to prove the termination of the first phase of CC.

Proposition 2. *Each node executes the* **EXIT** *command, i.e., terminates the cluster formation phase of CC.*

Proof. Let us call *mini-init* nodes those nodes that have the smallest weight in their neighborhood. We start by proving this result for the mini-init nodes.

Because of Proposition 1, all neighbors of a mini-init node v have assumed a role within a finite time. We distinguish two cases, depending on the role of v. If v is a clusterhead, not having smaller neighbors, immediately after having sent the CH message, it also sends an ASSOC_FINISH and exits (Algorithms 1, 7 and 8). If instead v is an ordinary node, after having decided its role it will receive, within a finite time, at least an ASSOC_FINISH message (the one of its clusterhead). Let now u be the last bigger neighbor of v that communicated its role via either an ASSOC_FINISH or JOIN message. When v receives one of these two messages from u, u has already decided its role, and therefore it executes the outermost else (Algorithms 7 and 8). Since u was the last node to communicate its role, all *Flag* variables at node v are **true** and therefore node v exits the protocol.

Let us now consider a non-mini-init node v. Since v's smaller neighbors have selected their role in finite time (Proposition 1), they have sent an ASSOC_FINISH or a JOIN message. (We note that this happens when v also has already selected its role.) In either cases, v executes the outermost else of the corresponding On Receiving procedures (Algorithms 7 and 8, respectively). Therefore, as soon as the last ASSOC_FINISH or JOIN message from its smaller neighbors, v exits the protocol. □

Proposition 3. *Each cluster is a clique.*

Proof. By cluster construction and from Algorithm 4. □

Corollary 1. *Each node belongs exactly to a cluster.*

Proof. Similarly to what shown in Lemma 1, once a node v has decided its role, i.e., it has set its variable *Clusterhead*(v) to either v (i.e., v is a clusterhead) or to $u \neq v$ (i.e., v is an ordinary node affiliated to u). Therefore, it cannot change its role anymore and at the same time, this is the only role it assumes. □

What shown above proves the following result.

Theorem 1. *The cluster formation phase terminates after having partitioned the networks into disjoint cliques each with one clusterhead and some ordinary nodes.*

3.2. Cluster interconnection

The cluster interconnection phase starts as soon as the cliques are formed. Given an initially connected network, a necessary and sufficient condition to obtain a connected backbone from the clusters produced by the previous phase is that of selecting paths between all pairs of clusterheads that are one, two or three hops away (the proof closely follows the lines of that in [18, Theorem 1]). For building these cluster connections each clusterhead needs to know all its "neighboring clusterheads", i.e., those clusterheads that are at most three hops away. Upon terminating the cluster formation phase, every node knows the ID and weight of each neighbor as well as the ID and the weight of the clusterhead to which each neighbor is affiliated. By having each node sending this information to its own clusterhead, we make the clusterheads aware of all the information they need to select paths for a connected backbone.

This information exchange is performed by using the following two new types of messages.

INFO(v,u) is sent from a node v to its clusterhead $u \neq v$ and contains the paths from v to other clusterheads that are at most two hops away (and hence at most three hops away from u). In order to optimize resources, instead of sending to u all possible ways to reach another clusterhead, v precomputes the best route to it (through *calculateBestRouteToOtherCluster*()). In this way, INFO packets are smaller, which implies lower energy consumption and a lesser likelihood of collisions.

SELECTION(C), where C is a set of nodes (those in a cluster) is a message broadcast by a clusterhead for informing its nodes about their role in the cluster, i.e., whether they are gateways or ordinary nodes. Through this message, routes are also communicated to all the nodes.

This phase of the CC protocol is started by each node executing the following algorithm.

Algorithm 9. InitBF {Executed by every node v}

```
if ($Ch(v)$ = false) then
    calculateBestRouteToOtherCluster();
    send INFO($v$, Clusterhead($v$));
```

Only ordinary nodes execute the body of the if in the procedure InitBF. It is for them to compute the best route to the clusterheads up to two hops away. The procedure *calculateBestRouteToOtherCluster*() works as follows. By knowing the IDs and weights of all its neighbors, and the IDs and weights of the neighbors' clusterheads, each ordinary node v computes routes to those clusterheads. If from v to any clusterhead there are multiple routes, one is chosen as follows: direct routes (one hop) are preferred. If this is not possible, among all two hops routes the one with the heaviest middle node is chosen. These information are coded in an INFO message which is unicast to the clusterhead.

Upon receiving an INFO message from one of its nodes u, a clusterhead v stores u's connection information (*store_connectionInfo*(u)). It then checks whether it has received all INFO messages. If this is the case, v can determine the "best paths" to all the clusterheads that are at most three hops away (Algorithm 10). This is done by executing the procedure *calculateRoutes*() that works as follows. Clusterhead v first executes the *calculateBestRouteToOtherCluster*() procedure to determine its own connection information. Then, it uses the information gathered via the INFO messages. More specifically, clusterheads that are one hop away are joined by their direct link. Clusterheads that are two hops away are connected via the common neighbor with the greatest weight (ties are broken based on the unique nodal ID). Finally, clusterheads that are three hops away are joined by selecting two *intermediate* nodes between them. The selection of these intermediate nodes between clusterheads c_1 and c_2, with $w_{c_1} > w_{c_2}$,[1] is performed as follows: among all nodes in c_2 cluster that are two hops away from c_1, the heaviest is selected. Among

[1] Without loss of generality we assume from now on that all nodal weights are different. If this would not be the case, as mentioned earlier, ties can be broken by using the unique nodal ID.

all its neighbors in c_1's cluster, the heaviest is selected. This procedure also sets Boolean variables such as *isGateway*(u) to **true** for every node u in v's cluster that has been chosen as a gateway, and to **false** otherwise. When a clusterhead v has calculated the routes to all the other clusterheads up to three hops away, it sends this information to the nodes in its cluster, via a SELECTION message. Therefore, each node knows if it is used as a gateway to interconnect adjacent clusters, and how to reach neighboring clusterheads.

Algorithm 10. On receiving INFO(u, v) {Clusterhead v receives an INFO message from its node u}

```
store_connectionInfo(u);
if (received INFO messages from all nodes in cluster)
  then
  calculateRoutes();
  send SELECTION(Cluster(v));
```

The procedure On receiving SELECTION(u), executed by a non-clusterhead node when it receives a SELECTION message from its clusterhead u, is very simple, and has node v updating some local variables about its role in the cluster and the role of the other cluster nodes, as well as all the routes selected by the clusterhead toward other clusterheads.

Example. The following example shows how a backbone is built from the cluster partition depicted in Fig. 1(b). All nodes execute procedure *InitBF*. Nodes 1 and 2, like 5 and 6, 13, 14 and 17 and 9 and 10 have no neighbors that are affiliated to a different clusterhead than their own. Each of them unicasts this information to its clusterhead. Node 4, affiliated to clusterhead 7, knows that it can reach clusterhead 3 directly. Node 4, therefore unicasts this information to its clusterhead. Node 8, affiliated to clusterhead 11, knows that it can reach clusterhead 7 directly, and that it can also reach clusterhead 15 through node 12 and clusterhead 18 through node 16. Node 8 unicasts this information to its clusterhead. Similarly, node 12, affiliated to clusterhead 15, and node 16, affiliated to clusterhead 18, know that they can reach clusterhead 11 through node 8. Each of them unicasts this information to its clusterhead. When clusterhead 3 receives INFO from all nodes in its cluster, it knows that the only reachable clusterhead is 7 through node 4. Therefore it unicasts this information to all nodes in its cluster via a SELECTION message. When clusterhead 7 receives INFO from all the nodes in its cluster, it knows, from INFO of node 4, that it can reach clusterheads 3. It also knows that it can reach clusterhead 11 through node 8. Node 7 unicasts this information to all nodes in its cluster. When clusterhead 11 receives INFO from all the nodes in its cluster, it knows, from INFO of node 8, that it can reach clusterhead 7 using the common node 8 as gateway and that it can reach clusterheads 15 and 18 using two intermediate gateways, nodes 8 and 12 for clusterhead 15, and nodes 8 and 16 for clusterhead 18. Node 11 unicasts this information to nodes in its cluster. Similarly, when clusterheads 15 and 18 receive all INFO from nodes in their cluster, they know, from INFO of node 12 and 16, respectively, that they can reach clusterhead 11 using node 12 for clusterhead 15, and node 16, for clusterhead 18. The result of the cluster interconnection on the clustering of Fig. 1(b) is shown in Fig. 1(c), where solid nodes and links are those in the backbone.

The correctness of this phase of CC is direct consequence of the assumption that all messages that are sent are correctly received, and of the proof that by connecting all clusterheads at most three hops away a connected backbone is obtained.

3.3. Backbone reorganization

What makes the CC solution most interesting is the ability of the nodes in the backbone to reorganize themselves quickly and efficiently when nodes either fail or are added. Here we detail the actions undertaken by a node after the removal or the addition of a neighboring node. (The actual "clique advantage" will be then made clear in the next section, where we quantify the difference between CC and previous solutions with respect to time and overhead).

In the description of the procedures we assume that a message sent by a node is received correctly within a finite time by all its neighbors. The information that each node knows is that known by a node at the end of the backbone formation (ID, weight and role of all neighbors, which nodes are in the clique and the ID and weight of the clusterheads its neighbors are affiliated to).

We use the same type of messages and notation used in the previous sections. Additionally, we use new types of messages that are needed for implementing the reorganization of the backbone.

3.3.1. Removing nodes

In the case nodes are removed from the network, the remaining nodes communicate by using messages such as CH, JOIN, INFO, INFO_REQUEST and DEAD. The first three messages have been introduced for cluster formation and interconnection. The two new types of messages are the following.

With an INFO_REQUEST(C) message, a clusterhead v sends to all the nodes in the set C a request for updated connection information. This information is needed to recompute routes to clusterheads up to three hops away. In other words, v asks each of the nodes in the set C to recompute the "best routes" to adjacent clusters, and to send this information back to it (similar to Algorithm 9). With this message the clusterhead also indicates the ID of the node that has been removed. In so doing, it asks to recompute only the routes that passed through that node. Notice that the node executing the procedure *calculateBestRouteToOtherCluster* try to find new routes to use instead of those passing through the node that has been removed.

The DEAD(u,C) message is used by a node v to inform each of the nodes in a subset C of its neighbors about node u removal (e.g., for death by energy depletion). The DEAD message is a unicast message.

Here are the operations of this phase of CC.

Algorithm 11. Link_failure(u) {Node v is made aware of the removal of node$u$}

```
update_Info(u);
if (u ∈ Cluster(v)) then
  Cluster(v): = Cluster(v)\{u};
if (Ch(u)) then
  for all (z ∈ Neigh(v):Clusterhead(z) = u) do
    Clusterhead(z) :=−1;
if (Ch(v)) then
  if (isGateway(u) or Ch(u)) then
    send INFO_REQUEST(Cluster(v));
else
  if ((Ch(u) or isGateway(u)) and (v and u are connected
    on the backbone)) then
    send DEAD(u, ConnectionNodes);
  if (Clusterhead(v) = u) then
    if ({∀z ∈ Cluster(v):$w_v > w_z$) then
      send CH(v);
      Ch(v) :=true;
      Clusterhead(v) :=v;
      send INFO_REQUEST(Cluster(v));
    else
      Clusterhead(v) :=−1;
```

When a node u is removed all its neighbors v are made aware of this (possibly by a service of the MAC layer, or because they have not received messages from u for more than a prescribed time). As a consequence, they execute the *Link_failure* procedure described in Algorithm 11. Node v first removes all the entries about u from its own data structures (*update_Info*(u);). If node u is in the same cluster of node v, v also removes node u from the list of nodes in its cluster.

If u was an ordinary node nothing more is needed, all other nodes maintain their role and all connections among clusterheads are still valid.

If u was a clusterhead, node v sets the value of *Clusterhead* of all its neighbors affiliated to u to -1.

The rest of the operations depends on whether v is a clusterhead or not. In the affirmative, v checks whether u was a gateway or a clusterhead itself. In this case, v requests to all the nodes in its cluster to send updated connection info, since, with u gone, connectivity might be lost. This is performed via broadcasting an INFO_REQUEST message. If v is not a clusterhead, but was connected to u and u was a backbone node (either a gateway or a clusterhead), then v has to communicate to all those neighbors in the backbone that were connected to u through v that u is no more. This is performed by sending (unicast) a DEAD message to each of the nodes in *ConnectionNodes*, the set of v's neighbors in the backbone that reached u through v. At this point v has only to check if u was its clusterhead. If this is the case, v and its fellow nodes have to find a new clusterhead. Node v takes a shot at it, and checks whether it is the big guy after u. In this case, it becomes the new clusterhead, set the appropriate variables, and asks its new nodes to update its connection information by broadcasting an INFO_REQUEST message. If v is not the heaviest node, it just sets its *Clusterhead* variable to -1 and wait to receive a CH message.

Algorithm 12. On receiving DEAD(u,z) {Node v receives from z news about u's removal}

```
update_Info(u);
if (Ch(u)) then
  for all (t ∈ Neigh(v):Clusterhead(t) = u) do
    Clusterhead(t) :=−1;
if (Ch(v)) then
  send INFO_REQUEST(Cluster(v));
else
  send DEAD(u,{Clusterhead(v)});
```

Upon receiving a DEAD message about the removal of a node u, every node v updates the information on u and if u was a clusterhead, sets the variable *Clusterhead* of all its neighbors affiliated with u to -1. If v is also a clusterhead, then it needs to recompute paths to neighboring clusterheads, and requests updated connectivity information to its nodes. Otherwise, v simply makes aware its own clusterhead that u is no more (by unicasting a DEAD message). The details are described in Algorithm 12.

When a node v receives an INFO_REQUEST message from its clusterhead (Algorithm 13) it knows that its clusterhead needs information to possibly reconnect the backbone. As a first thing, v's stores info about the node that has been removed, whose ID is part of the INFO_REQUEST message. This corresponds also to update the information about that node. Moreover, if the removed node was a clusterhead, node v also updates the information about its neighbors that were affiliated to it. Node v then checks if it has the information about the clusterhead of all its neighbors. If for some neighbors z *Clusterhead*(z) $= -1$, node v waits to hear news about z's new clusterhead before replying to its clusterhead. This is done by setting the Boolean variable have_to_send_info to **true**. In particular, have_to_send_info, initialized to **false**, is set to **true** only when a node receives an INFO_REQUEST message from its clusterhead and it is still waiting to receive information about its neighbors clusterheads.

When all information about v's neighbors clusterheads are updated node v recomputes the "best routes" to all the clusterheads that are at most two hops away formerly reached through the removed node. Then, it sends an INFO message to its own clusterhead bearing only these new routes. If the dead node is v's clusterhead then v sends *all* routes to its new clusterhead.

Algorithm 13. On receiving INFO_REQUEST(*Cluster*(u)) {Node v receives this message from its clusterhead u}

```
update_Info(dead_node)
if ($\forall z \in Neigh(v)$:$Clusterhead(z) \neq -1$ and $Clusterhead(v) \neq -1$) then
    calculateBestRouteToOtherCluster();
    send INFO($u$);
else
    have_to_send_info :=true;
```

When a node v receives a CH message from node u, updates the information pertaining u. Then, it checks if node u is in its own cluster. If so, u is the new clusterhead of v and node v becomes a member of u's cluster. As a consequence, v broadcasts to all its neighbors the corresponding JOIN message.

If node v was waiting to send an INFO message to its own clusterhead (have_to_send_info is **true**) and if it now knows about all its neighbors affiliation then it computes the new "best routes" to the clusterheads that are at most two hops away, and unicasts the corresponding INFO message to its clusterhead. The variable have_to_send_info is reset to **false**.

Algorithm 14. On receiving CH(u) {Node v receives CH from u}

```
Ch($u$) :=true;
update_Info($u$);
if ($u \in Cluster(v)$) then
    Clusterhead($v$) :=$u$;
    send JOIN($v,u$);
if ($\forall z \in Neigh(v)$:
$Clusterhead(z) \neq -1$ and have_to_send_info) then
    calculateBestRouteToOtherCluster();
    send INFO(Clusterhead($v$));
    have_to_send_info :=false;
```

When a node v receives a JOIN message with which node u communicates that it is joining node t's cluster, it first updates its information on u and on t. If it still have to send updated information to its clusterhead (have_to_send_info is **true**) and v knows about all its neighbors clusterheads, then it sends the corresponding INFO packet to its clusterhead after having recomputed the "best routes" to adjacent clusters. The variable have_to_send_info is reset to **false**.

Algorithm 15. On receiving JOIN(u,t) {Node v receives news that u is joining t's cluster}

```
update_Info($u$);
update_Info($t$);
if ($\forall z \in Neigh(v)$:$Clusterhead(z) \neq -1$ and Cluster-
    head($v$)$\neq -1$ and have_to_send_info) then
  calculateBestRouteToOtherCluster();
  send INFO(Clusterhead($v$));
  have_to_send_info :=false;
```

Once a clusterhead v has received all the INFO messages from its nodes it calculates new routes to the neighboring clusterheads. The procedure executed to receive INFO messages and perform the new route computation is similar to Algorithm 10. In this case, however, the computation of the new routes is optimized to broadcast to the cluster nodes only information about routes that have changed. The corresponding SELECTION message is therefore smaller.

The procedure On receiving SELECTION concerns only updating certain data structures about routes and roles at the cluster nodes.

3.3.2. Backbone reorganization correctness: Nodes removal

We start by assuming that nodal removals happen far enough in time that the network can converge to a connected backbone before it has to deal with another removal. In the following we also discuss how the protocol procedures can be extended in case of multiple "overlapping" node removals.

Proposition 4. *Whenever a node removal occurs, the CC protocol reorganizes the backbone in finite time. By the end of the reorganization process*:

1. *Each node affected by the removal has deleted from its data structures all the information regarding the removed node. In other words, no node is affiliated to a node that has been removed, no node believes to have such node in its cluster, or believes to be able to route packets through a removed node.*
2. *Each node has assumed either the role of clusterhead or of ordinary node in exactly one cluster.*
3. *Each cluster has exactly one clusterhead, and satisfies the clique property.*
4. *The backbone resulting from the reorganization is connected (provided the network topology still is).*

Proof. To prove this proposition we consider separately the cases corresponding to the removal of an ordinary node, of a gateway, and of a clusterhead.

- *Case A*: Removal of an ordinary node v. All v's neighbors execute the Link_failure(v) procedure (Algorithm 11), removing information on node v from their list of neighbors (*update_Info*(u)). Nodes belonging to v's cluster also remove v from the list of nodes in their cluster. No other operation is performed and no message is exchanged. Since node v was not used to relay packets for other clusters (i.e., it was not a gateway), nodes outside its neighborhood do not maintain any information about v in their data structure. The operations performed by executing Algorithm 11 serve the sole purpose of deleting all entries related to node v (property 1). At the end of the execution of Algorithm 11 by v's neighbors properties 2 and 3 are also satisfied: all the remaining nodes have maintained their role, being either ordinary nodes or clusterheads in a single cluster. Furthermore, clusters still satisfy the clique property: A clique is still a clique when one of its nodes is removed. Finally, since each pair of nodes x and y was connected (before v's failure) via paths which did not include node v, and since each remaining node is still serving in the same role, and acting as gateway between the same pairs of clusters, the backbone resulting at the end of the reorganization is still connected. In fact the backbone is still the same, i.e., it is unaffected by the removal of an ordinary node.

- *Case B*: Removal of a gateway node v. All the neighbors of v execute the Link_Failure(v) procedure (Algorithm 11), removing information on node v from their data structures (*update_Info*(u)). Nodes in v' cluster also delete v from the list of nodes in their cluster. All the gateways which are neighbors of v in the backbone inform the adjacent gateways which were using the path through v as well as their clusterheads that v has been removed by broadcasting a DEAD message (Algorithms 11 and 12). This makes adjacent clusterheads aware of the possibility to elect new gateways and update the routes to v's cluster. This is made possible by having the clusterheads in adjacent clusters sending INFO_REQUEST messages to their nodes (Algorithm 12) and by v's clusterhead doing the same (Algorithm 11). As in the cluster interconnection phase, upon receiving an INFO_REQUEST message, the nodes in a cluster recompute their routes to the adjacent clusters and unicast this information (via an INFO message) to their clusterhead. The clusterhead in turn will make a final decision on the routes, informing the new gateway nodes by means of SELECT messages. This is exactly what happens in case of a reorganization upon a gateway failure. Termination is an immediate consequence of the nature of the messages exchanged. Whenever a node receives a message it can immediately answer it providing the requested information. Since all the neighbors of v and the backbone nodes belonging to adjacent clusters that used node v for interconnection (i.e., all and only the nodes which had maintained an entry for node v) cancel information on v (*update_Info*(u), Algorithms 11 and 12), property 1 is met at the end of the reorganization. As in case A, no node changes its role. The only changes that happen concern the nodes now entering the backbone for interconnecting their cluster to some of the adjacent ones. Properties 2 and 3 are thus also met at the end of the reorganization. We are left to prove that the backbone resulting from the reorganization is still connected (provided that the network topology still is). The connectivity of the backbone is a direct consequence of the rules used for cluster interconnection. Let us consider any two nodes in the network, node x and node y. If the network topology is connected after v's removal then there exists at least a path $P{:}x = v_1, v_2, \ldots, v_k = y$ between x and y. Let us now consider any two adjacent nodes in P, say v_i and v_{i+1}. In the hierarchical organization formed by CC they either belong to the same cluster or to different clusters. In the first case they are directly connected. In the second, since v_i and v_{i+1} are adjacent they must belong to adjacent clusters. This means that their clusterheads are at most three hops apart, and that there is a path on the backbone that interconnects the two nodes.
- *Case C*: A clusterhead node v is removed. As for the previous cases all v's neighbors execute the Link_Failure(v) procedure (Algorithm 11), removing information on node v from their data structures (*update_Info*(u)). Nodes in v's cluster also delete v from the list of nodes in the cluster. DEAD messages (Algorithms 11 and 12) are sent over the backbone so that all adjacent clusterheads are made aware of v's removal. This in turn triggers requests by such clusterheads to their nodes to recompute the routes toward the cluster where v was, once a new clusterhead has been elected. The initial partitioning of nodes into disjoint clusters is maintained (i.e., nodes that were in v's cluster will still be part of a cluster including the remaining nodes). However, a new clusterhead will be elected, and adjacent clusters will have to be informed of the identity of the new clusterhead. The election of the new clusterhead is performed by executing Algorithm 11, and it is weight based. Upon detecting node v removal, all nodes in v's cluster set their variable *Clusterhead* to -1 (as the identity of their clusterhead is currently undefined). The node with the biggest weight, however, elects itself as the new clusterhead, and broadcast the corresponding CH message. As soon as the nodes in its cluster receive this message, they answer with a JOIN, re-affiliating to the cluster, and acknowledging the clusterhead election. Gateways selection to interconnect the "new" cluster with the adjacent ones is then performed similarly to Case B. The new clusterhead (Algorithm 11) as well as the clusterhead of adjacent clusters (as soon as they detect that v is gone, Algorithm 12) issue INFO_REQUEST messages asking their nodes to recompute the routes to adjacent clusters. Once such information is gathered the clusterheads make decisions on the way to interconnect to adjacent clusters and gateways are instructed (via a SELECT messages) about the interconnections they have to take care of. The only difference from the cluster interconnection phase is that when a node receives an INFO_REQUEST (Algorithm 13) it

may still be unaware of the role of some of its neighbors (the ones with the *Clusterhead* variable set to −1). Nodes therefore set a variable have_to_send info to 1, and delay sending the INFO message to when all the neighbors have decided their role. Since role decision corresponds to sending either a CH or a JOIN message, whenever a CH or a JOIN message is received nodes verify whether they now know the role of all their neighbors. If this is the case, they can compute the routes having a complete picture of their neighborhood (Algorithms 13, 14 and 15), they do so, and send the routing information to their clusterhead. Termination is straightforward. A new clusterhead is selected and its neighbors are informed about it in finite time, given that the messages that are sent are immediately answered. This also means that nodes receiving a CH or JOIN message sent by a node z will set back *Clusterhead*(z) to a value different from −1. In turn, this means that in finite time all nodes will be able to answer the clusterheads INFO_REQUEST messages with the corresponding INFO messages, and that in finite time the backbone will be reorganized. Property 1 holds because of arguments similar to those in cases A and B above: All the nodes that maintain entries associated to node v in their data structures are informed of node v removal via DEAD messages or because, being v's neighbors, they realize that v is gone directly. Therefore, the entry corresponding to v are correctly deleted. The clusters obtained at the end of the reorganization still satisfy properties 2 and 3, since nothing has changed but the identity of the clusterhead formerly lead by v. Backbone connectivity (property 4) can be proven as in case B.

Therefore, independently of the node that is removed, CC is always able to converge to a connected backbone. □

The case with multiple removals can be dealt with similarly. The procedures of this phase of the protocol always lead to a connected backbone. Only the performance of the protocol is affected. For instance, if multiple ordinary nodes are removed, each of them can be dealt with at the same time as single removals (Case A of the proof above), according to the order in which a node realizes that their ordinary neighbors are gone. The protocol actions are slightly more involved in case multiple gateways are removed in close succession. In this case, responding to the first removal of a gateway, clusterheads close by receive DEAD messages that trigger INFO_REQUEST messages and the corresponding computation of new routes. Chances are that the new routes pass through a node that has been removed in the meanwhile. This last removal, however, will trigger the execution of the Link_failure procedure (Algorithm 11), which eventually will ask for a new route computation. A similar reasoning applies to when multiple clusterheads go. Actions such as the selection of a new clusterhead in a cluster, the following cascade of JOIN messages, the local update of the data structures as well as the setting of the *Clusterhead* variables to −1 happens in parallel with no concern. A possible wrong or inefficient route could be generated at the time of the DEAD messages "storm" followed by that of the INFO_REQUESTs, etc. Eventually, however, successive iterations triggered by realizing that after a clusterhead, another has been removed, etc., will lead to converge to a connected backbone. Mixed situations are also likely to arise, where ordinary nodes, but especially gateway and clusterheads could be removed in close time proximity. These situations can be dealt as before. Overall, concurrent removals are likely to impose higher overhead, and longer convergence times, because of the higher traffic generated, as well as because, for this same reasons, collisions can more likely occur. However, as far as a CC message is received at a node within finite time of its transmission, the protocol is bound to repair the backbone after the removal of any kind of its nodes, provided, of course, that the network per se remains connected (A lengthier formal proof can be found in [19]).

Example. Let us consider the backbone depicted in Fig. 1(c). Let us assume that node 1 fails. Node 2 and clusterhead 3 are made aware of the failure, and remove 1 from the list of nodes in their cluster. Node 1 is an ordinary node so no other reorganization or information exchange is needed. The result of the reorganization procedure is depicted in Fig. 2(a). Node 3's cluster is still a clique and the backbone is still connected. Assume now that clusterhead 18 fails. In this case nodes 17 and 16 are made aware of the failure, and remove node 18 from the list of node in their cluster. Nodes 17 and 16 then need to check if they have to forward the failure information. Node 17, not being a backbone node does not have to forward any information. However, node 16 needs to send a DEAD message to node 8 because it was used with node 8 to connect clusterheads 11 and 18. At this time each node checks if it is the better node in the cluster for becoming a clusterhead. Node 16 knows that node 17 is better and therefore does not promote itself to clusterhead. Node 17 instead knows that it will become the new clusterhead and broadcasts this information via a CH message asking also for connection information with a INFO_REQUEST message. When node 8 receives the DEAD message from node 16 it updates its information about node 18, forwards this information to its clusterhead 11 and locally records that node 16 must choose its clusterhead. When node 16 receives the CH message sent by 17 it replies broadcasting a JOIN message to made aware all neighbors about its new clusterhead. When it receives the INFO_REQUEST message it replies with an INFO message saying that it can reach clusterhead 11 through node 8. When node 11 receives the DEAD message sent by 8 it updates its information about node 18 and sends an INFO_REQUEST to nodes in its cluster to try to find a route to the new clusterhead. Node 8, after the reception of the JOIN message sent by node 16, knows the new clusterhead of node 16 and can reply to its clusterhead saying that it can

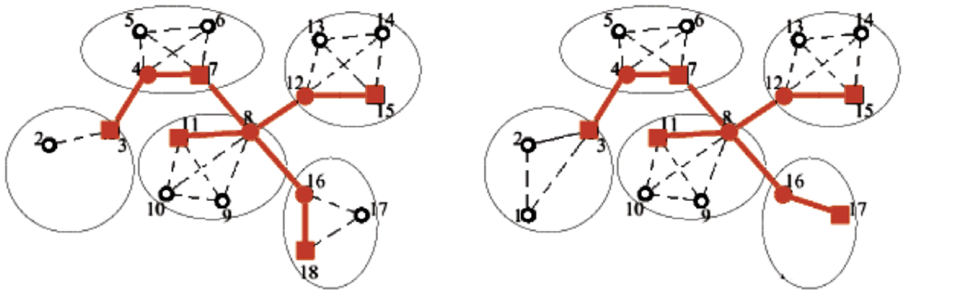

(a) Node 1 fails: Nothing changes
(b) Clusterhead 18 fails: Node 17 is the new clusterhead

Fig. 2. Backbone reorganization.

reach the new clusterhead 17 through node 16. Now, clusterheads 11 and 17 know how to reach each other. At the end of the reorganization procedures the new clique is created and the backbone is still connected (Fig. 2(b)). Only local exchange of information are needed to react to a clusterhead failure. The case of gateway failure is treated similarly to the failure of a clusterhead. In this case, it is not needed to find a new clusterhead, and only the INFO_REQUEST and INFO messages are exchanged to find routes to connect the clusterheads.

3.3.3. Adding new nodes

In case new nodes are added to the network they need to be included into some existing clusters or they will have to form a new cluster. Furthermore, the backbone needs to be extended to include, or cover, the new arrivals.

The following description refers to the addition to one node to the network. We discuss later on how to deal with arrivals *en masse*. The types of messages used in this phase of the CC protocols are CH, JOIN, INFO, INFO_REQUEST, BEACON, STATUS and NEW_NODE. The first four message types have been introduced for previous phases. The three new types of messages are the following.

BEACON(n) is sent by a new node n to inform the surrounding nodes about its presence. This is a broadcast message.

STATUS(v,n) is sent by a node v to a node n after the reception of BEACON(n). This message is used to inform n about v's role and affiliation. If v is a clusterhead, it contains v's weight and the list of nodes in v's cluster. If v is not a clusterhead, this message carries v's weight, and the ID and weight of v's clusterhead. This is a unicast message from node v to node n.

NEW_NODE(*Clusterhead*(v), n) is sent by a node v to its clusterhead to inform it about the presence of the new node n. This is a unicast message from node v to its clusterhead.

When a node n is added to the network it broadcasts a BEACON message. The message includes n's own ID and weight.

When a node v receives the BEACON message from n, it gets to know the ID and weight of the new node. At this time (Algorithm 16) it replies with a STATUS message.

Algorithm 16. On receiving BEACON(n) {Node v hears a BEACON from n}

```
update_Info($n$);
if ($Ch(v)$ = true) then
    STATUS ← cluster nodes;
    STATUS ← $w_v$;
else
    STATUS ← $w_v$;
    STATUS ← $w_{Clusterhead(v)}$;
    STATUS ← $Clusterhead(v)$;
send STATUS($v$,$n$);
```

Algorithm 17. On receiving STATUS(u) {Node n receives STATUS from u}

```
store_statusInfo($u$);
if (received status information from all neighbors) then
    Decide_Role();
```

In particular, after having stored information about the new node, if v is a clusterhead, it packs the STATUS message body with information about its cluster nodes and its weight. If v is not a clusterhead, it forms the packet with its own weight, and the weight and ID of its own clusterhead. The message is then unicast to n.

Once node n has received a STATUS message (Algorithm 17), it stores the info for node u (*store_statusInfo*(u)).

When node n has received a STATUS message from all its neighbors, it has all the information for deciding its own role and the cluster to join (or to create).[2] This is performed by executing the *Decide_Role*() procedure (Algorithm 18). Node n starts by checking if it can affiliate to a neighboring clusterhead, if any, without disrupting the clique property of that cluster. If this can be done, node n joins the bigger cluster. It then informs all its neighbors of its new role by

[2] For a new node, the problem of getting to discover *all* its neighbors is non-trivial. From a practical point of view, the initial discovery period could be defined by means of a time-out mechanism, after which the node acts based on the information it has gathered.

broadcasting a JOIN message. If node n cannot join any cluster, it becomes a clusterhead and creates it own cluster (with only one node, itself). Therefore, it broadcasts a CH message to its neighbors and calculates the "best routes" to the clusterheads one or two hops away.

Algorithm 18. Decide_Role() {Used by a new node n to decide role and cluster}

```
if (association to some cluster is possible) then
    $x$ :=$BestClusterhead()$;
    $Clusterhead(n)$ :=$x$;
    send JOIN($n$,$x$);
else
    send CH($n$);
    $Ch$(n) :=true;
    $Clusterhead(n)$ :=$n$;
    $calculateRoutes()$;
```

Algorithm 19. On receiving JOIN(u,t) {Node v receives JOIN from u}

```
$Clusterhead(u)$ :=$t$;
$update\_Info(u)$;
$update\_Info(t)$;
if ($u = n$)
    if ($Ch(v)$= true) then
        if ($t = v$) then
            send INFO_REQUEST($u$);
        else
            $calculateBestRouteToOtherCluster()$;
            send SELECTION($Cluster(v)$);
else
    if ($Clusterhead(v) \neq t$) then
        send NEW_NODE($Clusterhead(v)$,$n$);
```

Algorithm 20. On receiving NEW_NODE(v,n) {Clusterhead v receives NEW_NODE from u about the new node n}

```
if ($v$ does not know $u$) then
    send INFO_REQUEST($Cluster(v)$);
```

Algorithm 21. On receiving CH(u) {Node v receives CH from u}

```
$update\_Info(u)$;
$Ch(u)$ :=true;
$Clusterhead(u)$ :=$u$;
if ($u = n$)
    if ($Ch(v)$= false) then
        send NEW_NODE($Clusterhead(v)$, $n$);
    else
        $calculateRoutes()$;
        send SELECTION($Cluster(v)$);
```

When a node v receives a JOIN message it executes Algorithm 19. At first, it updates the information about the sender and the sender's clusterhead. Further actions are needed only if the sender of the JOIN message is a new node n. In this case, if v is a clusterhead it first checks whether the new node is affiliating to it. If so, node v inquires the new node for routes to adjacent clusters via an INFO_REQUEST. In case v is a clusterhead and n affiliate to a different cluster, node v uses *calculateBestRouteToOtherCluster*() as described above for the case of the removal of a node. If executing this procedure v finds better routes through the new node, the nodes in v's cluster are informed via a SELECTION message. Finally, if v is an ordinary node, it informs its own clusterhead of the presence of the new node by unicasting the message NEW_NODE(*Clusterhead*(v),n), unless u is joining its own cluster.

When a clusterhead v receives a NEW_NODE message from a node in its cluster about a new node n that it does not know yet it sends an INFO_REQUEST message (like those seen for node removal) to all nodes in its cluster. As before, this is needed for updating or for setting routes to clusterheads up to three hops away that could be (better) reachable through n (Algorithm 20).

Node v executes Algorithm 21 when it receives a CH message. The node starts by updating the information about the sender. The only actions that need happening are related to when the new node is a clusterhead. Therefore, v checks whether the sender is the new node n. In the positive, if node v is not a clusterhead, then it unicasts a NEW_NODE message to its clusterhead to inform it about the presence of the new node and the possibility to set up new routes. If node v is instead a clusterhead, then it just needs to update its routes and send those routes that pass through the new node to its nodes (via a SELECTION message). We note that in this case, differently from the case of cluster interconnection and node removal, v does not need to receive INFO messages from its nodes to gather new connection information, since the new clusterhead is just one hop away from it. Therefore, v is itself the preferred node to interconnect v's cluster to n's cluster.

3.3.4. Backbone reorganization correctness: Addition of nodes

The proposition below proves the correctness of this phase of the CC algorithm when one node at a time is added to the network (We discuss the case of multiple nodes entering the network below.).

Proposition 5. *Whenever a new node n enters the network, the CC protocol reorganizes the backbone in finite time. By the end of the reorganization process*:

1. *Each node affected by n's presence has stored n's information in its own data structures.*
2. *Each node has either the role of clusterhead or of ordinary node in exactly one cluster.*

3. *Each cluster has exactly one clusterhead, and satisfies the clique property.*
4. *The backbone resulting from the reorganization is connected.*

Proof. The nodes affected by the presence of the new node n are its immediate neighbors and, if they are ordinary nodes, their clusterheads. The first property is straightforward to achieve. The new node broadcasts a BEACON message that is received in finite time by its neighbors. Since in the first statement executed in every message-triggered procedure concerns storing information about a node, the reception of the BEACON message enables the neighboring nodes of the new nodes to store its information. Since in finite time the new node decides its role (see below), it sends either a JOIN or a CH message, which triggers its neighbors to send a NEW_NODE message to their clusterheads. This passes the information about the new node n also to its neighboring clusterheads, proving property 1.

Whenever a new node n is added to the network it BEACONs its presence. The nodes z that receive the BEACON message, in response, send back their role, affiliation, and possibly (if they are clusterheads) their nodes (Algorithm 16). Based on this information node n can locally decide whether to affiliate to a cluster (and to which one) or to become clusterhead itself (Algorithms 17 and 18). All this happens in finite time. When n's neighbors receive information on the role taken by n they tell to their clusterheads that a new node has arrived (via a NEW_NODE message, in Algorithms 19 and 21). This in turn triggers a recomputation of the routes to the cluster to which n affiliated, or that it created. More precisely, if n has become a clusterhead, adjacent clusters have to compute routes to its clusters. If n has joined an existing cluster there could be "better" routes to that cluster via n. Nodes in clusters adjacent to n receive from their clusterhead an INFO_REQUEST message asking them to recompute the routes to n's cluster and send the INFO back to their clusterhead which will make a final decision on the routes and will instruct the gateways, as usual, via a SELECTION message. The clusterhead node n affiliated to also asks n for routes to the adjacent clusters to see whether the routes to such clusters can be improved via n. In case n is itself a clusterhead, it can locally compute such routes. Moreover, after having received status information on all its neighbors node n will execute the procedure *Decide_Role*(). Therefore, the new node always assumes a role. This proves property 2. The same argument also proves property 3, since if a new cluster is formed by the new node, this cluster only has a clusterhead as member (n itself, which is a clique), and all other clusters in the network stay untouched, preserving the clique property. If n decides to join a cluster, it does so only if it has found an existing cluster of all members of which n is a neighbor. That cluster is therefore still a clique, and it has only one clusterhead (the one it had before).

Finally, property 4 is a consequence of Algorithms 19 and 21, where a clusterhead always computes either new routes or "better routes" through n to neighboring clusterheads or to n according to the criteria that maintain the backbone connected (the principles of [18] are implemented in procedures *calculateBestRouteToOtherCluster*() and *calculateRoutes*()).

The termination of the reorganization process is no concern: Only messages that can be directly answered are exchanged at all times, and since every message is always received in finite time as it was sent, every node is able to act according to its message-triggered procedures. □

While for multiple removals the CC algorithms are able to converge to a connected backbone, the case with arrivals en masse (i.e., when multiple nodes join the network at the same time) require some more sophistication than what provided by the procedures in this section. There are no problems in case the addition of nodes happens so that the two node are four or more hops away. Given that the arrival of a new node affects only its one hop neighbors and their clusterheads, the corresponding additions are independent, and are easily treated as single ones. The problem of multiple additions around the same nodes concerns assuring the maintenance of the clique property. With the current procedures, for instance, two nodes that are added basically at the same time and that are not neighbors of each other, might be however neighbor of the same clusterhead v, to whom they both could affiliate. Clearly, if they both decide to affiliate with v, chances are that the first new node thinks it can safely join v's cluster, and sends its JOIN message to v (and other surrounding nodes) after the second new node has joined v's cluster already. This could compromise the clique property. The solution to this potential flaw requires the introduction of additional types of messages, and of some extra procedures. The idea is that of letting the clusterhead to have the final say on the joining of extra nodes. The new arrivals should seek permission from the clusterhead before affiliating to its cluster. In this way, each clusterhead will check whether a new node can safely be added to its cluster at the time of request. If it can invite the node maintaining its cluster as a clique, node v will send an acknowledgment to the requesting node, and will add the node in its cluster. Only after receiving such acknowledgment, a node will send a JOIN message informing its neighbors of its new role as an ordinary node under v.

Example. Let us consider the backbone depicted in Fig. 1(c). Let us now consider the appearance of node 19 as shown in Fig. 3(a). Node 19 broadcast a BEACON message that is received by node 1 and 3. Node 1 is not a clusterhead and so it replies communicating to 19 its role and weight and the ID and weight of its clusterhead. Node 3 is a clusterhead and therefore it replies to 19 communicating its weight and the list of nodes in its cluster. When node 19 receives the information from nodes 1 and 3 it decides its own role and cluster. According to what received from node

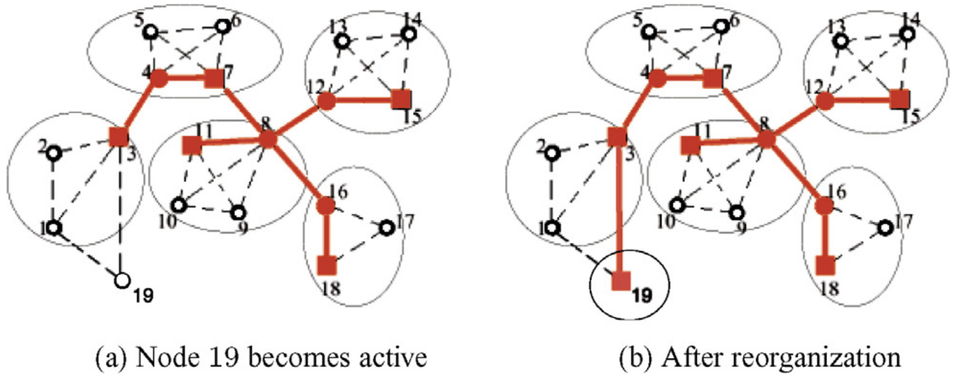


Fig. 3. Backbone reorganization with new node: New cluster.

3, node 19 knows that it can not be part of 3's cluster, because it cannot communicate directly to node 2. Thus node 19 decides to become clusterhead (sending a CH message). It creates it own cluster and calculates the routes to reach other clusterheads. Node 19 knows that it can reach only clusterhead 3 with a direct link. When node 1 receives the CH message from node 19 it sends a NEW_ NODE message to its clusterhead communicating the presence of the new node. When node 3 receives the CH message from node 19 it knows that it can get to node 19 via a direct link. Therefore, it updates the information about the routes to reach other clusterheads and sends this information to nodes 1 and 2. The result of the reorganization procedure is depicted in Fig. 3(b). The clusters in the network are still cliques and the backbone is still connected. Assume now that node 20 becomes active as shown in Fig. 4(a). Node 20 broadcasts a BEACON message which is received by its neighbors 1, 2, 3, 9 and 11. Nodes 1, 2 and 9 are not clusterheads and so they reply communicating their information to 20. Nodes 3 and 11 are clusterheads and therefore they reply to 20 communicating their weight and the list of nodes in their clusters. When node 20 receives the information from these nodes it realizes that it can be part of 3's cluster and that it cannot be part of 11's. As a consequence, node 20 broadcasts the JOIN message to its neighbors. When node 1 and 2 receive the JOIN message from node 20 they do not send the NEW_NODE message to their clusterhead 3, because 20 is associating to node 3. When node 3 receives the JOIN message from node 20 it sends a message of INFO_REQUEST to 20 to update its connection information. When node 9 receives the JOIN message from node 20 it sends a NEW_NODE message to its clusterhead communicating the presence of the new node. When node 11 receives the JOIN message from node 20 it knows that it has to update its connection information and thus calculates the new best routes. It then sends a SELECTION message to the nodes in its cluster. When node 11 receives the NEW_NODE message from 9 it checks if the message is about a node already known and thus ignores it, otherwise node 11 knows that has to update its connection info and sends a message of INFO_REQUEST to node in its cluster. When node 3 receives the INFO message from node 20 it knows that it can reach clusterhead 11 using 20 as gateway. Node 11 knows that it can reach clusterhead 3 using 20 as gateway. The result of the reorganization procedure is depicted in Fig.4(b). The clusters in the network are still cliques and the backbone is still connected.

4. DMAC

The Distributed and Mobility Adaptive Clustering (DMAC) protocol was introduced in [6,20] for providing a distributed and localized clustering algorithm that would be resilient to nodal mobility in wireless ad hoc networks. An extension of DMAC for WSNs, called Sensor-DMAC

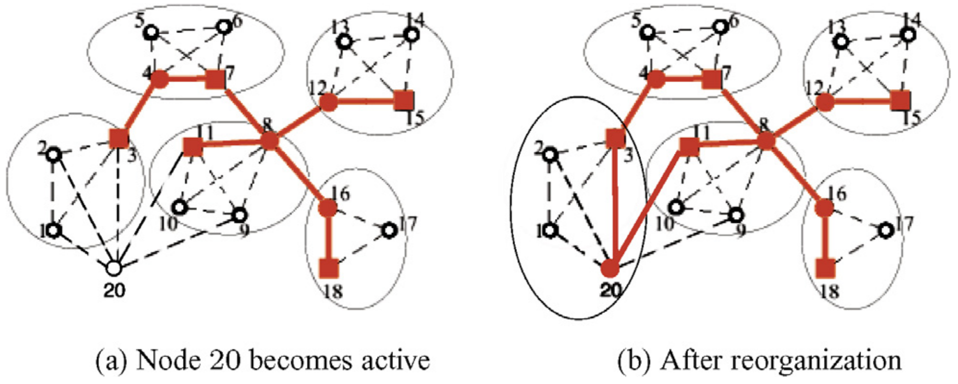


Fig. 4. Backbone reorganization with new node: Association to a cluster.

[5], was later proposed for topology control, where the salient features of DMAC were used to select nodes that could "stay awake" in the network without compromising connectivity and network functions.

We describe the protocol as executed by the generic node v with weight $w_v > 0$, similar to CC (In case two nodes have the same weight, ties are broken by using the unique nodal ID.). In the following, communications among neighboring nodes are considered to be either beacon-based, i.e., local broadcasts from a node to all its neighbors, or unicast, where a packet is intended for one specific destination.

Upon entering the protocol, node v starts listening in *promiscuous mode*, i.e., to all the packets that are sent, even if those packets are not addressed to it. In doing so for a prescribed period of time τ, v becomes aware of its one hop neighbors, and their weights. Based on this information, v is able to decide whether it will be a DMAC clusterhead or an ordinary node. The "decision rule" is similar to that of CC's: *A node v decides after all heavier neighbors have decided.* More precisely, if there are no heavier neighbors around v, v becomes a clusterhead, and broadcasts this information to all its neighbors (using a CH message). As for CC, in this case v is an init node. If instead v has some heavier neighbors that are clusterheads, it will affiliate with the heaviest one, once all the neighbors have broadcast what they have decided (by using a CH or a JOIN message).

By the end of this first phase (cluster formation) of DMAC a node has been assigned a role, which is either clusterhead or ordinary node. In the latter case, the ordinary node is affiliated to the heaviest clusterhead in its neighborhood. By this time a node also knows the ID, weight and role of all its neighbors, and, if these are ordinary nodes, it knows ID and weight of their clusterheads. A clusterhead that has received connection information from the nodes in its cluster chooses the gateways to its neighboring clusterheads (DMAC second phase: Cluster interconnection). Gateways are selected so that there is a path between every pair of clusterheads that are two or three hops away [18], similarly to the CC cluster interconnection (Section 3.2). Differently for CC's clique clustering, however, there cannot be two clusterheads that are one hop apart.

DMAC clustering formation and interconnection have been intensely investigated in the past. For details, and experimental evaluation, the reader is referred to [6,20,21,7]. In this paper we add to the implementation of these two phases the implementation of the DMAC third phase, which, similarly to CC's, deals with node removal and addition. In particular, we used the same messages and procedures that we defined for CC in order to form a DMAC connected backbone. The messages are INFO, SELECTION and INFO_REQUEST, here modified to carry the connection information that are needed specifically for DMAC. This information is usually lighter than the corresponding one in CC messages because all communications in DMAC clustering pass through the clusterhead. Messages of type DEAD, BEACON and NEW_NODE are also used to report the removal and addition of a node, similar to their use in CC.[3]

5. Experimental results

We investigate and compare the performance of CC and DMAC via simulations. More specifically, we implemented the protocols using the VINT project network simulator (ns2) [22].

5.1. Simulation scenarios, metrics and experiments

Our implementation is based on the CMU wireless extension to ns2, i.e., on the IEEE 802.11 MAC with the DCF. Some of the standard parameters of 802.11 have been modified to take sensor nodes characteristics into account. These include shorter transmission radius and different values for the energy model. These values come from the actual sensor prototypes developed within the IST Energy Efficient Sensor Networks (EYES) project [23].

The sensor deployment area is a square of side L where N static wireless sensor nodes are randomly and uniformly scattered. The transmission radius of each node is set to 35 meters. Two nodes are neighbors if and only if their Euclidean distance is less than or equal to the transmission radius. In our simulations, the number of nodes N varies among the values 50, 100, 150, 200, 300, and 400, while L has been set to 200 m. This allows us to test our protocol on increasingly dense networks, from (moderately) sparse networks with average degree equal to 4 to dense networks where the average degree is around 33.5.

Our comparative performance evaluation proceeds in three steps.

In our first group of experiments we focus on the network set up, i.e., on the backbone formation. We assume that the N nodes are scattered randomly and uniformly on the deployment area and run the first two phases of the CC protocol and the clustering formation and interconnection procedures of DMAC. Our investigation aims at quantifying the time and overhead associated to backbone formation, as well as the distinctive differences between the properties of backbones generated by CC and those built by DMAC.

Our second set of experiments concerns the effectiveness of CC and DMAC in reacting to nodal removal. Nodes are removed randomly and uniformly from networks where an initial clustering formation and interconnection have been performed until the network itself disconnects (This hap-

[3] The original definition of DMAC [6] did not consider explicitly the removal and addition of a node. A function from the MAC layer was supposed to communicate that the link to a neighbor was no longer available or that a new link was established to a newcomer. Here we explicitly discover a missing neighbor or a new node via BEACON messages.

pens, on average, after 35–50% of the nodes have been removed from networks with 50–400 nodes, respectively.).

The third step of experiments is devoted to comparing the performance of CC and DMAC in case nodes are added to the network over time. We create connected backbones where only 80% of the N nodes are deployed. Then, once a backbone has been set up, we add the remaining 20% of the nodes one at the time. The 20% of the N nodes are chosen randomly and uniformly from the original batch, and their addition to the network is also random.

In our experiments, we consider the following metrics:

- *Protocol duration*, i.e., the time needed by each protocol to complete its operations. In particular, we have considered both the time needed for backbone formation and the time needed for backbone reorganization after a node is removed or added. The contribution of each phase of the protocol to the overall protocol duration has been quantified.
- *The overhead* (in bytes) associated to the protocol operations. These are physical layer measurements, which account for collisions and for the corresponding automatic packet retransmissions at the MAC layer.
- *Nodal involvement*, i.e., the number of nodes involved in cluster and backbone reorganization as a consequence of the removal or addition of a node.
- *Backbone size*, i.e., the number of the network nodes that are in the backbone. We have studied how many of the network nodes are clusterheads and how many are instead introduced into the backbone for the sole purpose of clusters interconnection (i.e, how many are gateways).
- *Route length*, i.e., the average among all the shortest path lengths, over the subgraph G_b of the network topology graph induced by the backbone construction. In DMAC the subgraph G_b is the graph where the only active links are those that form the backbone, and those between ordinary nodes and their clusterhead. In CC the subgraph G_b is the graph made up of those links that form the backbone and those between the nodes in the same clique. This metric gives a measure of how fast and effectively (also in terms of energy since for short range technologies the power consumed to transmit over a link is basically constant) packets can be communicated over the source-destination path.

Our performance evaluation results have a 95% confidence and a 5% precision.

5.2. Backbone formation: Setting the network up

In this section we focus on the first two phases of CC and DMAC, namely, clustering formation and interconnection. Overall output of these two phases is a connected backbone used for network operations. Upon completion of the cluster interconnection phase, the network "is up."

Fig. 5(a) shows the average size of the backbones generated by CC and by DMAC. CC produces bigger and denser backbones (up to 78% bigger than DMACs in networks with 400 nodes). The reason is that, having to meet the clique property, clusters tend to be smaller. The average size of a CC cluster is 8 when $N = 400$ while it is 17 for DMAC clusters. The resulting larger number of clusters induces a larger number of CC clusterheads (up to 110% higher than for DMAC) and a higher number of adjacent clusters to which each small clique cluster has to interconnect (and therefore a higher number of gateways), as showed in Fig. 5(b) and in Fig. 5(c). Starting from similar values when $N = 50$, the number of adjacent clusters of a CC and DMAC cluster significantly differ for bigger networks. When $N = 400$ the number of CC adjacent clusters is 82% higher than that for DMAC, resulting in more clusters being interconnected and into a more meshed backbone.

Understanding why the difference in size between CC and DMAC backbones is smaller than the difference in the number of their clusterheads gives us further insight on the design of efficient backbone formation protocols. The number of nodes included in a backbone depends on the kind of gateways used for interconnecting adjacent clusters. In CC adjacent clusters can be frequently joined through their clusterheads (one hop connectivity), which never happens for DMAC, where the set of the clusterheads form an independent set over the network topology graph. In Fig. 6 we show the percentage of one hop, two hops and three hops interconnections in networks with 300 nodes (we observed that increasing or decreasing the number of nodes has little impact on such percentages.) We found out that 30% of the CC intercluster connections are directly through the clusterheads (no extra node added to the backbone); 50% of the CC connections require only a single gateway, common to the clusterheads of which it is neighbor; finally, only 20% of the connections require a pair of intermediate gateways. In a DMAC backbone clusterheads can never be directly joined. Two third of the adjacent clusters can be interconnected via common gateways, and the remaining one third of the DMAC interconnections is made up of a pair of intermediate gateways. This explains the fact that the average number of nodes entering the backbone as gateways that enable the connection of a pair of clusterheads is higher in DMAC than in CC.

Since nodes in a clique cluster can directly communicate and given that the CC backbone is denser, route lengths are shorter in CC than in DMAC. The average route length on CC backbones are up to 25% shorter than the routes that traverse a DMAC backbone (see Fig. 7).

It is interesting to investigate in depth the distribution of the size of the clique clusters for increasing network density, and how it differs from that of DMAC (see Fig. 8). Since in DMAC each clusterhead invites all its neighbors to its cluster, clusters of clusterheads with a very high nodal degree have a very big size. This is confirmed by the obser-

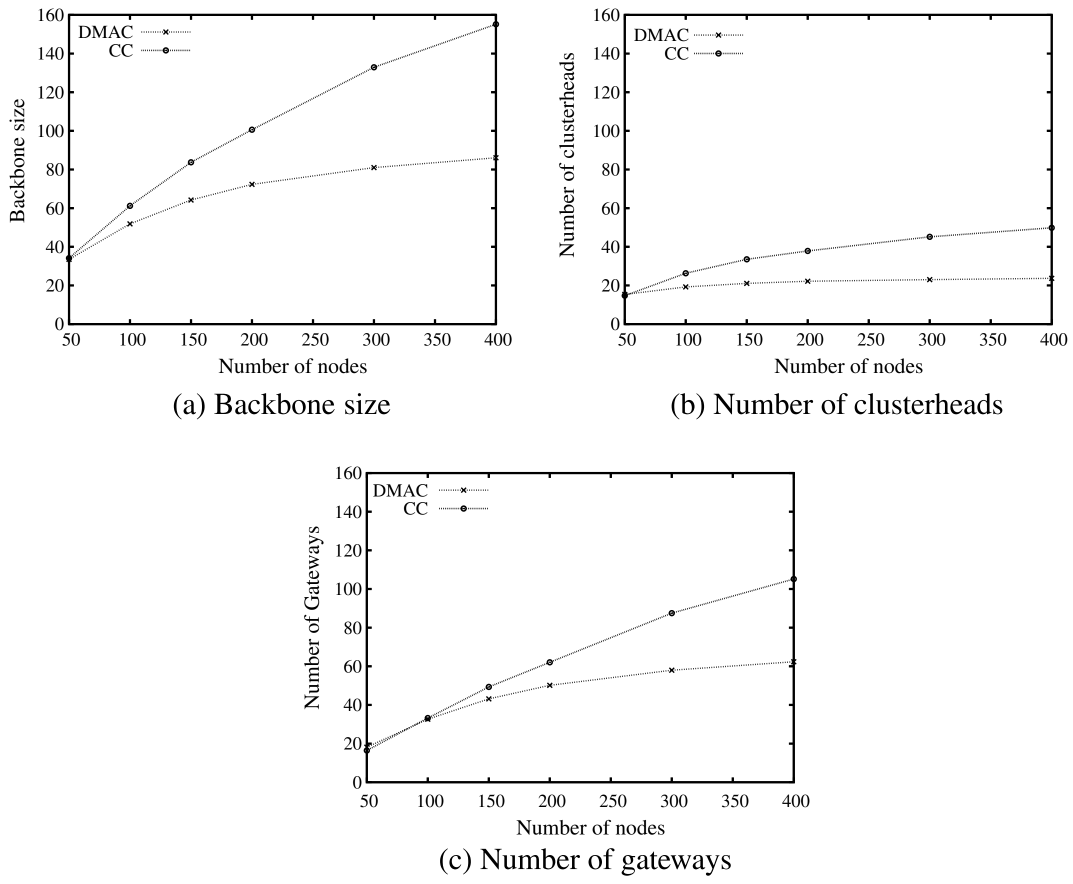


Fig. 5. Backbone metrics: size = clusterheads + gateways.

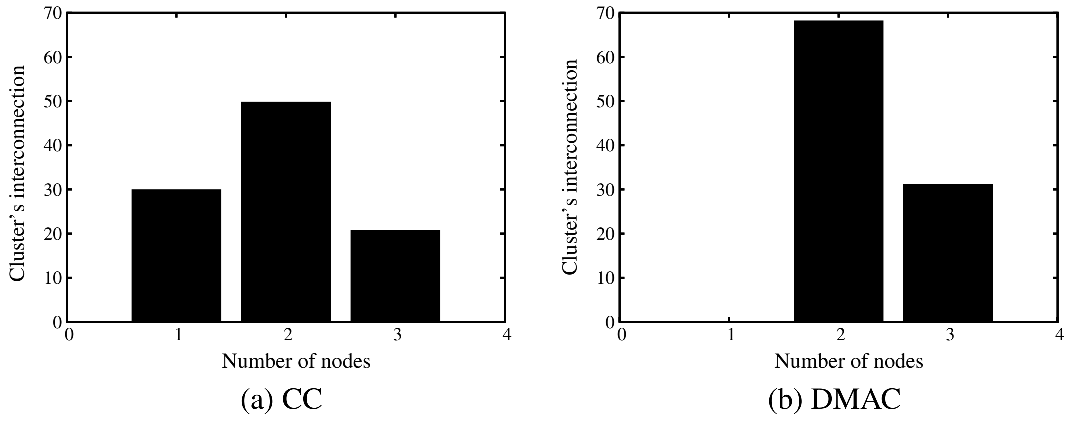


Fig. 6. Percentage of one, two and three hops cluster interconnections.

vation that the clusters size distribution reflects the distribution of the nodal degree. Because of the clique property, invitation in a CC cluster are more selective. However, at the considered realistic densities the majority of the cliques have sizes ranging from 4 to 10/12 nodes. These are reasonably high sizes for clusters that have to be cliques.

We now dig into two important metrics: The time and overhead required by the two protocols for forming a backbone.

Fig. 9(c) shows the number of bytes exchanged on average for backbone formation. A first observation is that this overhead is non-negligible compared to what needed to react to node removal/addition. In order to minimize the overhead associated to backbone formation and maintenance it is therefore important to properly and carefully design protocols for backbone formation and for backbone reorganization. In terms of overhead of the formation phase one would expect the overhead to be much higher

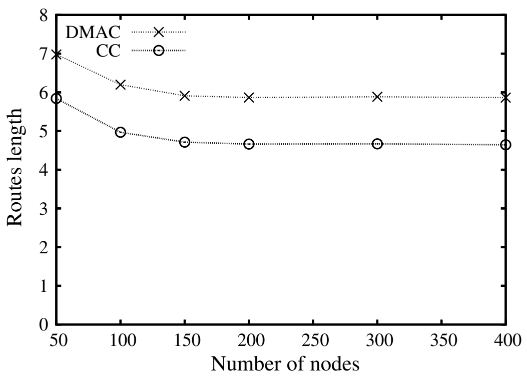


Fig. 7. CC vs. DMAC: Average backbone route length.

in CC. Partitioning into cliques requires the exchange of more information (e.g., the list of the neighbors of a neighbor), and therefore higher overhead. Moreover, during the phase of cluster interconnection more information has to be exchanged to make the nodes aware of the routes to follow for sending packets to a different cluster. DMAC nodes only refer to their clusterhead. This is the only node that needs to know to which gateway the packet has to be relayed for reaching an adjacent cluster. Each CC node, instead, can directly relay a packet to the gateway and therefore needs to be informed of which node is serving as gateway for which interconnection. Optimizations can be performed in the protocol implementation to reduce the neighbors lists exchange, which is what we have done with both CC and DMAC. However, there is no way to avoid that CC needs to exchange more information during the cluster interconnection phase. This is reflected in the results for this phase which shows that the increase in overhead can be as high as 170% in CC with respect to DMAC. Possible protocol optimization can be achieved from a deeper investigation of which are the messages which contribute the most to the overall overhead (separate curves for each of the most overhead producing information exchanges are shown in Fig. 9(a) and (b). Our aim here is to understand which are the most critical phases for each of the two protocols.

In the backbone formation phase CC pays the price of messages which are not used in DMAC (e.g., NEIGHBOR, ASSOC_REQUEST and ASSOC_FINISH). In addition, more INFO messages are transmitted (given the need to interconnect to a higher number of adjacent clusters). Finally, the need to inform nodes about the routes to all adjacent clusters (since all nodes can directly communicate to the gateways) motivates a very significant increase in the overhead for exchanging SELECTION messages with respect to DMAC (see Fig. 9(b)). Overall, this leads to up to a 170% overhead increase in the second phase of the protocol. However, the most significant overhead is paid, in both protocols, during nodes partitioning into clusters. If we dig into the messages exchanged during this phase, CC exchanges, as expected, more CH messages than DMAC (there is a higher number of clusterheads), and less JOIN messages (because there are less ordinary nodes). While the overhead associated to these messages is non negligible (especially for the JOIN messages) the three fourth of the overhead experienced during the first phase of the protocols is due to the exchange of many small acknowledgment messages needed to confirm the reception of CH and JOIN messages.[4] Although this is not something that is evident at the level of protocol description, this is an important part of the implementation of this kind of protocols and needs to be taken into account when designing the protocols themselves. Overall, the first phase (clustering formation, see Fig. 9(a)) is the phase with the highest message exchange (more than twice the information required for cluster interconnection). Message exchange optimization is needed to bring CC to considerably reduce the increase in overhead vs. that of DMAC during this phase. The overhead paid for partitioning into cliques is 25% higher than what needed for DMAC clustering. The overall overhead paid by CC for backbone formation is 41% higher in CC than in DMAC. We will see that this additional overhead (imposed by the information that needs to be exchanged for cluster formation, and hence unavoidable) pays off during the network operations, where having clusters that are cliques leads to remarkable savings in the overhead induced by backbone maintenance. When considering together backbone formation and maintenance the CC definitely leads the race in key metrics. For instance, the total overhead observed through the experiment duration is 235% higher in DMAC than in CC (see Section 5.3).

In terms of time (and maximum time) needed for backbone set up both protocols show similar performance. The average time needed to partition nodes into clusters and that needed for cluster interconnection varies from 2.5 s ($N = 50$) to 15–18 s, depending on the protocol, when $N = 400$. As expected, the higher the number of nodes and the network density, the higher the number of messages that need to be exchanged, the longer the time needed to complete the backbone formation. The dominant component of the backbone formation time is the time needed for the initial clustering (first phase). For both protocols the second phase (cluster interconnection) never lasts more than 1.6s. The worst case scenarios (maximum times) are not remarkably different from the average case: in all our runs the backbone formation has been completed in less than 23 s.

5.3. Maintaining the backbone: Dealing with node removals

In this section we present the results of the set of experiments devoted to assessing the effectiveness of CC in dealing with the removal of network nodes. We start from a fully formed backbone, built on top of a flat network of N nodes, and we start removing nodes one by one

[4] The implementation of all messages in our experiments is acknowledgment-based. Upon receiving either a broadcast or unicast message, every node replies with an "ack" of a very few bytes.

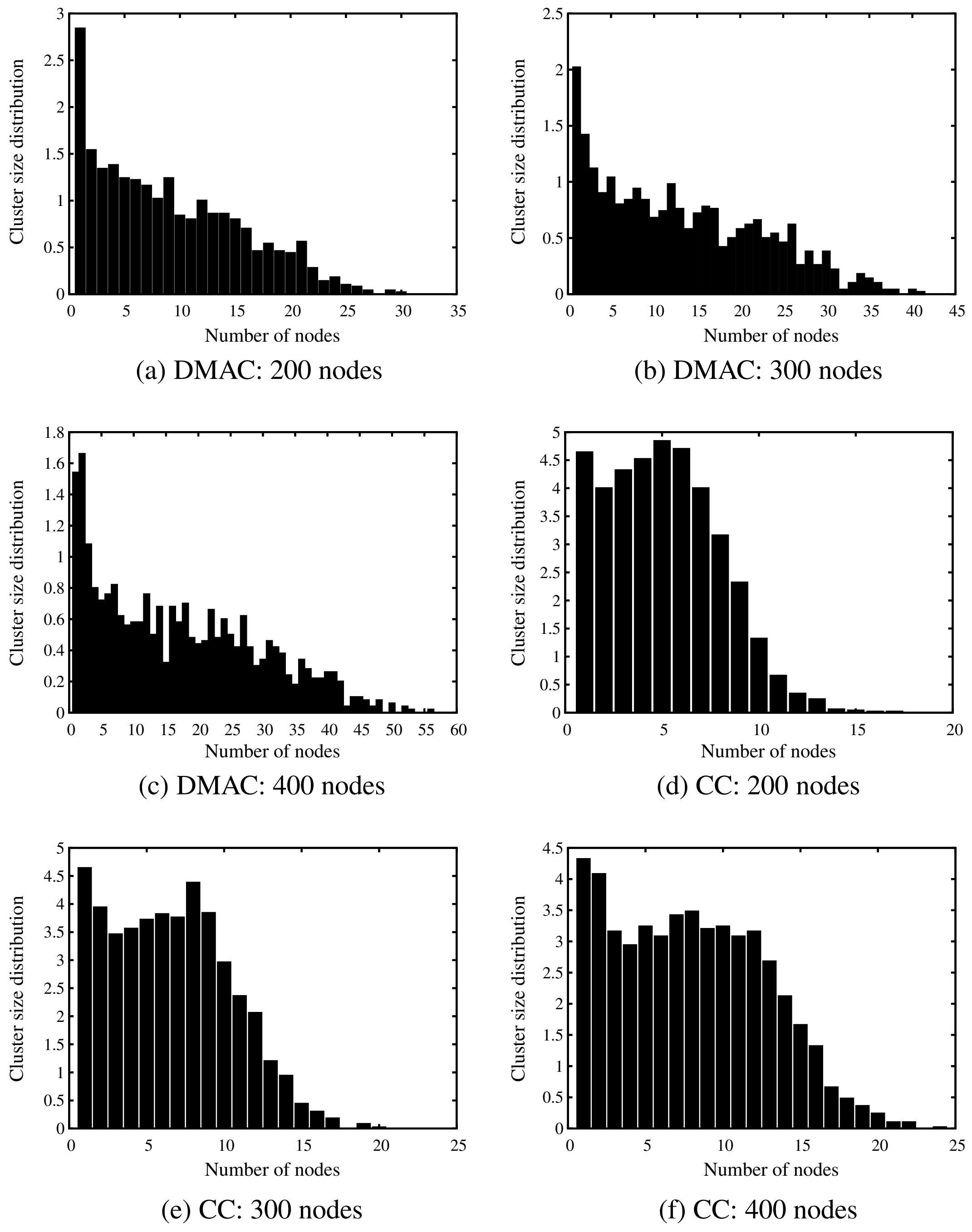
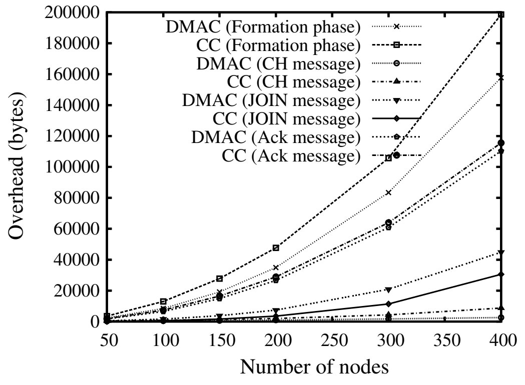


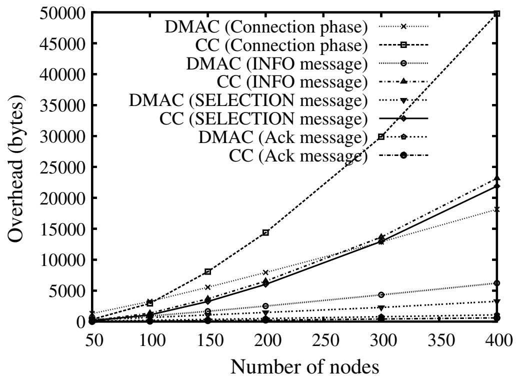
Fig. 8. Cluster size distributions.

randomly and uniformly, until the network is disconnected, i.e., when it is, in general, no longer able to perform its functions. Each run of our experiments simulates how CC and DMAC deal with such a sequence of removals.

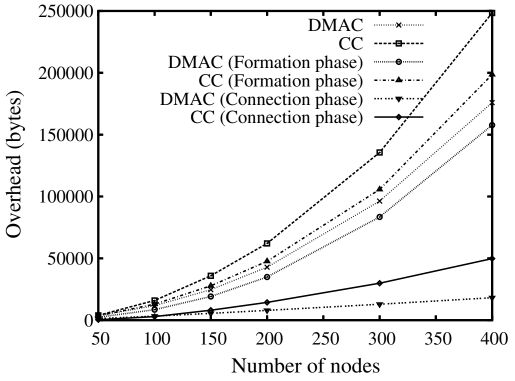
Results are presented in Figs. 10–13. Fig. 10 shows the average time needed to CC and DMAC to reorganize the backbone after a node has been removed (both average and worst case). The figure clearly shows that CC achieves a significant improvement. In networks with 400 nodes DMAC needs 63% more time than CC to converge to a connected backbone after a removal. This difference is even more evident if we consider the maximum reorganization duration. CC reorganization always terminates within 4 s, while DMAC can last up to 30 s! This is due to the CC design choice of minimizing role changes and involving in the reorganization only nodes in the proximity of the removed one. This is not the case for DMAC, where the removal of a node can trigger a chain of role changes that could involve nodes far away in the network. This leads to nodes assuming different roles and affiliating to different clusters over time before the backbone stabilizes. We have quantified this differences in Fig. 11(a), which depicts the



(a) Overhead for cluster formation

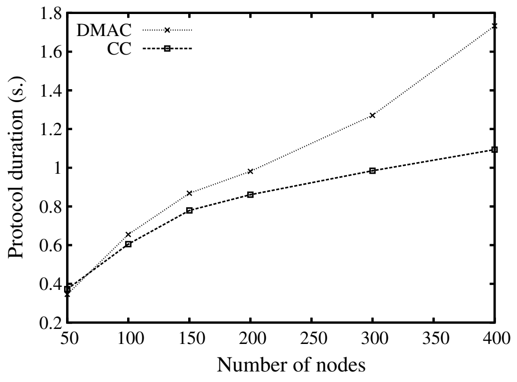


(b) Overhead for cluster interconnection

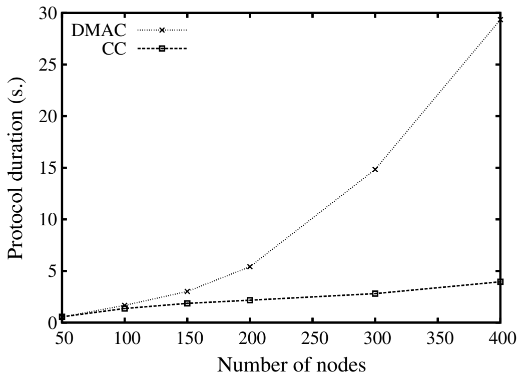


(c) Overhead for backbone formation

Fig. 9. Overhead (bytes) for the different phases of backbone formation.



(a) Average case

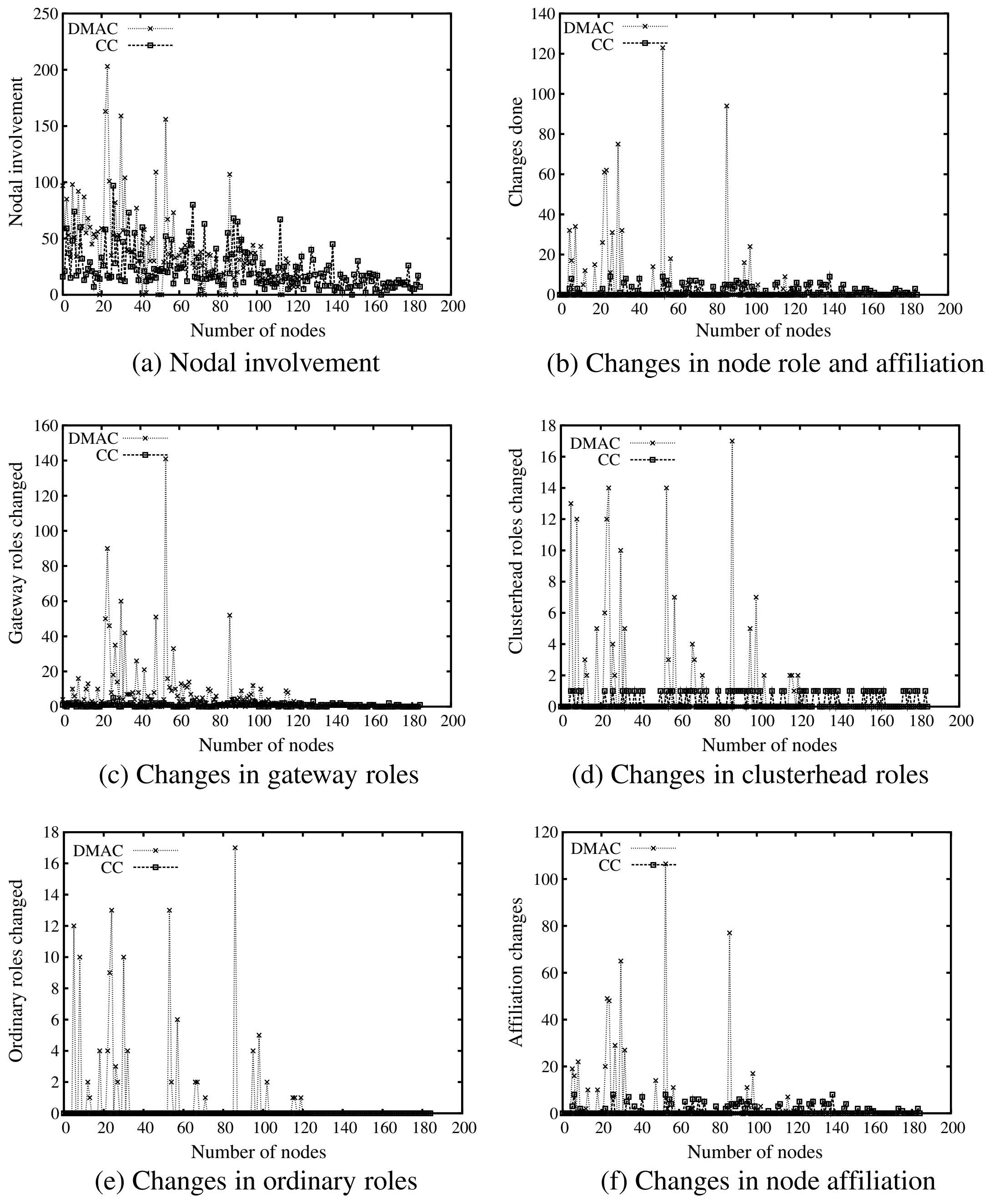


(b) Worst case

Fig. 10. Reorganization time (s) after the removal of a node.

number of nodes that are involved in a network reorganization (i.e., that exchange messages because of node removal), for a specific simulation run (in network with 400 nodes).

Each failure is represented by one point in the picture. The metrics are measured only for the removal of backbone nodes, since ordinary nodes removal is dealt with trivially by both protocols. The curves clearly show that DMAC experiences a high variance in the number of nodes involved in the backbone reorganization. When specific events occur, such as the removal of a heavy clusterhead v with a large number of heavy nodes in its cluster, it is highly likely that role changes and message exchanges are propagated far away from the node removed, involving big parts of the network. In this case, v's removal frees many heavy nodes that most likely will declare themselves clusterheads, gathering in their cluster nodes that were ordinary nodes, gateways and clusterheads in other clusters. This (and especially the affiliation of other smaller clusterheads) in turn leads to further changes that are



(a) Nodal involvement (b) Changes in node role and affiliation

(c) Changes in gateway roles (d) Changes in clusterhead roles

(e) Changes in ordinary roles (f) Changes in node affiliation

Fig. 11. Nodal involvement and changes in roles and affiliations after a node failure in networks with 400 nodes.

needed to reorganize the backbone: Multiple chain reactions are started. This is shown by the peaks in the figure: 25–50% of the nodes can be involved in the reorganization. When averaging over all node removals the nodes involved in the reorganization are about 10% of all the network nodes in DMAC, while only 5% of them react to a nodal removal in CC.

In order to better understand what happens in these critical situations we have investigated the number of changes that occur in the roles assumed by nodes (clusterhead, ordinary node, gateway), and in the nodes cluster identities during backbone reorganization (Fig. 11(b)). This captures the dynamics of the reorganization. If two protocols that involve a comparable number of nodes differ significantly in this metric it means that in one protocol nodes tend to change their role and affiliation multiple times while the backbone is being maintained. This is exactly the case for DMAC. While the change of roles and affiliations is limited to a few in CC (the worst case is when a clusterhead node fails, out of its cluster nodes one has to be elected clusterhead while the others re-affiliate to it), in DMAC the chain of reactions that can be triggered by a removal can lead to and order of magnitude higher number of role and affiliation changes. We have investigated the types of changes that most frequently occur. For the same run of simulations Fig. 11(c) displays the number of times a node enters

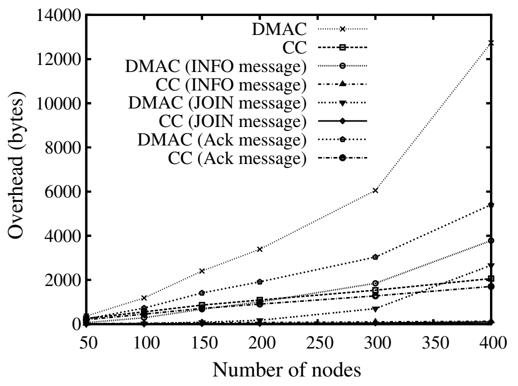


Fig. 12. Overhead (bytes) for backbone reorganization (cluster reorganization and interconnection).

the backbone as gateway or a gateway node exits the backbone while reorganizing upon a failure. Fig. 11(d) focuses on the number of times an ordinary node becomes a clusterhead (bounded by 1 in CC). Fig. 11(e) displays the number of times a clusterhead becomes an ordinary node during the network reorganization (this can never occur in CC). Finally, Fig. 11(f) shows the number of times a node changes its clusterhead during the network reorganization. Higher dynamics is experienced in the nodes affiliation to clusters. Moreover, DMAC gateways change more often. This is due to the described domino effect typical of this protocol, which leads to a chain of changes in cluster memberships. In CC the number of changes at any level is really kept to a minimum, explaining why CC is largely superior to DMAC in terms of overhead.

The longer time needed for backbone maintenance is not the only drawback of using DMAC. This protocol also imposes higher overhead for backbone maintenance (Fig. 12). After a removal the number of bytes that need to be transmitted by DMAC nodes is almost seven times bigger than that needed by CC in networks with 400 nodes. The reason is twofold. On one side, DMAC involves more nodes in the reorganization. On the other side, in DMAC it may happen that messages are exchanged to reflect a role change that is only temporary (e.g., a node that proposes itself as clusterhead might later realize that there is a better candidate to serve in that role). Exchanging only local information, and given that each node has a correct local view of the network, and that it makes stable decisions on role changes explain why CC shows lower overhead after a removal.

We have gone further than this. We have also investigated which are those, among the protocol operations that generate the highest overhead. We observe that the greatest transmission of bytes, on both protocols, happens because of INFO and JOIN messages, as well as to the need to acknowledge the protocol messages. The number of bytes transmitted for affiliating to a cluster is one order of magnitude lower in CC than in DMAC. In the former, only nodes in the cluster of the removed node can re-affiliate to a new clusterhead. In DMAC, the backbone structure can dynamically change several times during the reorganization. The number of connection information (INFO messages) which need to be exchanged is also significantly lower in CC than in DMAC. The reason is that the CC backbone structure is pretty stable. A node has to compute new routes only if there have been changes that could have compromised previously available routes. This seldom happens in CC. In addition, only the minimum number of nodes is involved in routes recomputation.

The rest of our experiments concerns the structure of the backbone generated and reorganized by the two protocols. Metrics concerning backbone characteristics have been discussed in detail in Section 5.2. Here we only briefly overview them.

Fig. 13(a) and (b) compare the size of the backbones generated by CC and DMAC on network topologies with a number of nodes varying between 50 and 400, as well as the route lengths on the generated backbones. Fig. 13(a) clearly shows the drawbacks of a clique-based approach. Having to meet the clique property clusters in CC tend to be smaller. Partitioning the nodes in smaller clusters, CC also needs a higher number of gateways to interconnect each cluster with all the adjacent ones. Being the number of clusters and the number of gateways higher, CC produces an overall bigger backbone than DMAC. On the other hand, the possibility to communicate directly between nodes belonging to the same clique and the high

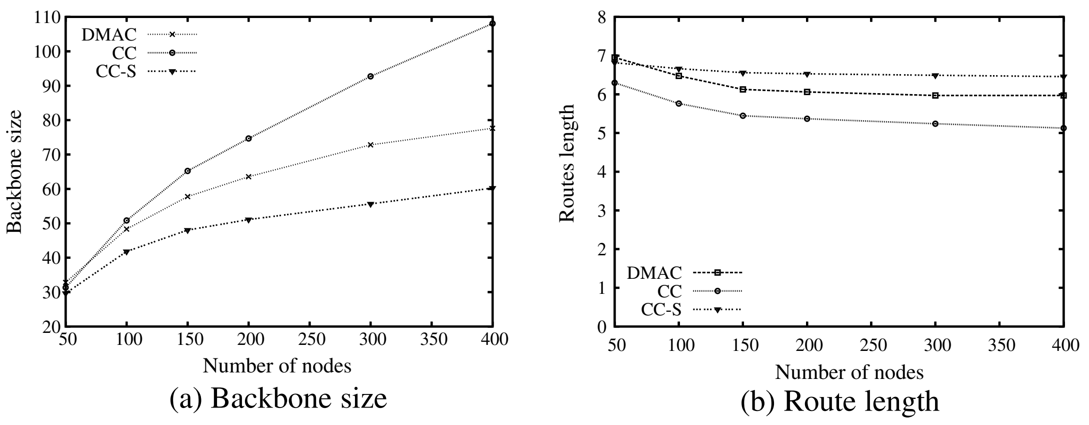


Fig. 13. Backbone size and route length after node removal.

density of the resulting backbone imply shorter routes in a CC backbone. The backbone to whom the maintenance operations (third phase of CC) converge to has very similar features to the one constructed during the first two CC phases. Therefore, for the characteristics shown so far, CC turns out to be a promising solutions for organizing a WSN hierarchically.

CC outperforms DMAC in all investigated metrics. Its only drawback might be the increased backbone size. When minimizing this metric is particularly important, especially in those WSN applications where the smaller the backbone the least the energy consumed [5], the CC backbone can be made smaller by breaking some of its (small) cycles. This, of course, would lead to a backbone that is still connected. Here is how, with breaking small cycles, we obtained a smaller backbone. We have previously proposed an efficient general method for backbone "sparsification" [7] that works like this. We define a *macro-link* as a path interconnecting two adjacent clusterheads and we consider the graph G_s whose nodes are the clusterheads and whose links are the macro-links. In order to get a sparser backbone the nodes in G_s exchange with their neighbors information on their macro-links so that triangles in G_s can be identified and broken (clearly maintaining the backbone connectivity). The criterion used to break a triangle favors deleting macro-links which traverse a higher number of gateways. Whenever all macro-links traversing a gateway have been deleted the gateway can be removed from the backbone. In our experiments we have also applied this technique to CC. We refer to this variant as CC-S. Preliminary results show that it is possible to get backbones that are up to 23% smaller than DMAC backbones maintaining the low overhead and fast reorganization features of the original CC protocol. Protocol operations are always completed within 4 s, and the resulting overhead is less than half what needed by DMAC despite the additional cost required to identify and break triangles. As expected, this comes at the price of longer route lengths (8% longer than DMAC).

5.4. Maintaining the backbone: Addition of a node

In this section we illustrate and discuss the results of our third set of experiments. As mentioned, we start with a connected backbone built off of 80% of the N network nodes. The remaining 20% are then added, one by one, every 30 s. Newly arriving nodes are randomly placed in the network. Their ID and weight are also randomly selected in a given range. In this scenario we investigate the time and overhead needed for backbone reorganization, as well as the properties of the resulting backbone after a node is added.

Our results, shown from Figs. 14–18, assess the advantages that the clique property brings in case of node addition. The overhead paid by CC for reorganizing the network

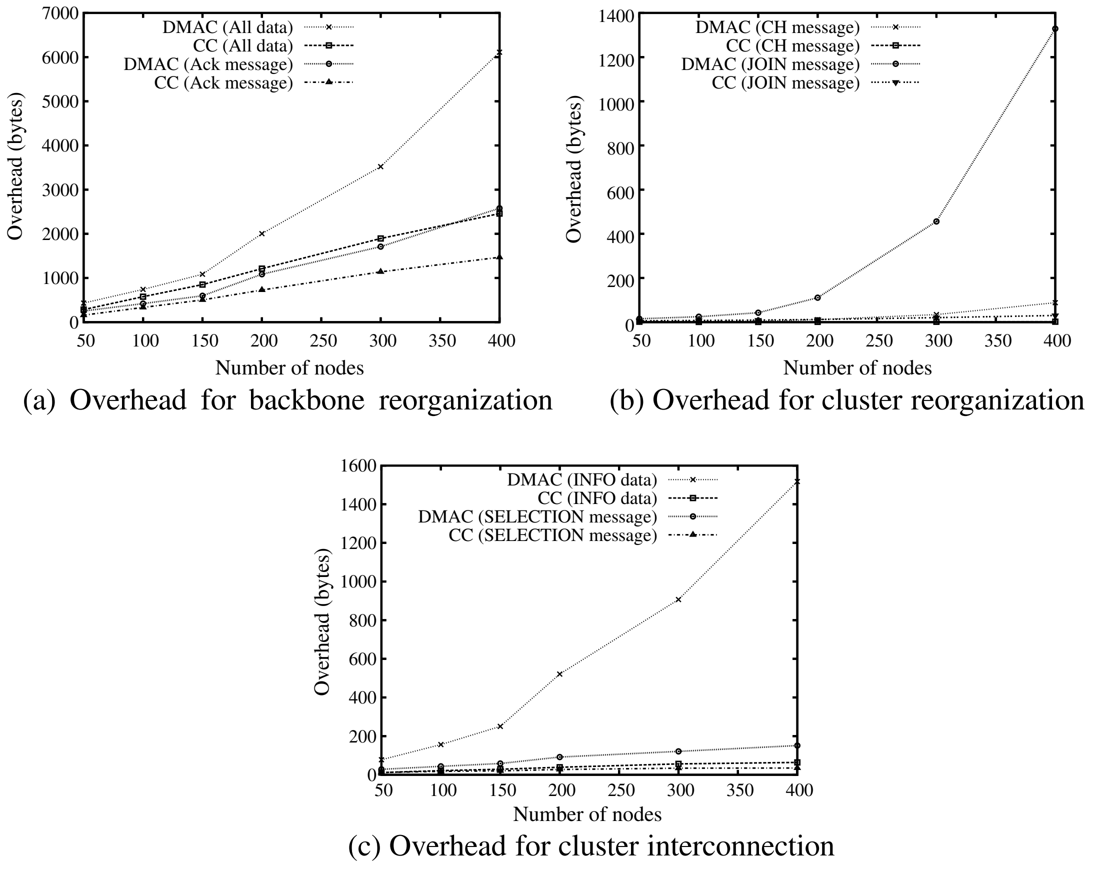


Fig. 14. Overhead (bytes) for backbone reorganization and cluster interconnection.

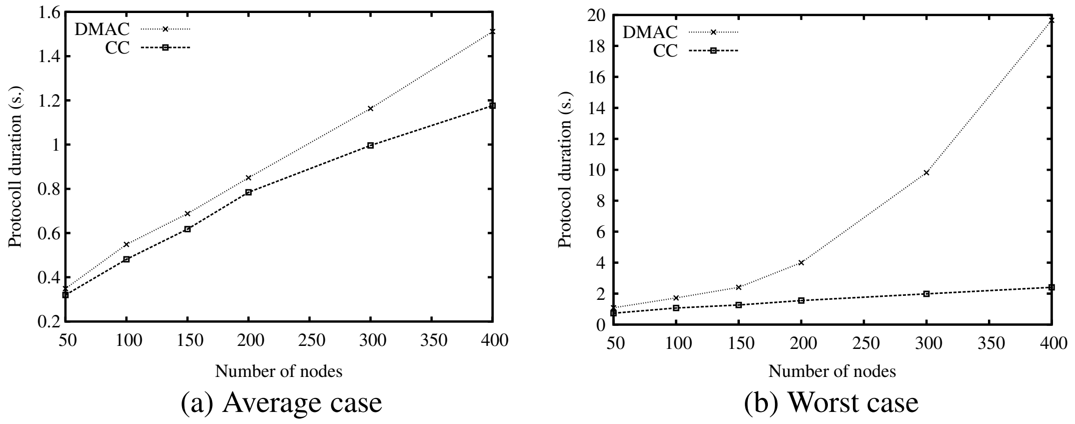


Fig. 15. Reorganization time (s) after a new node arrives.

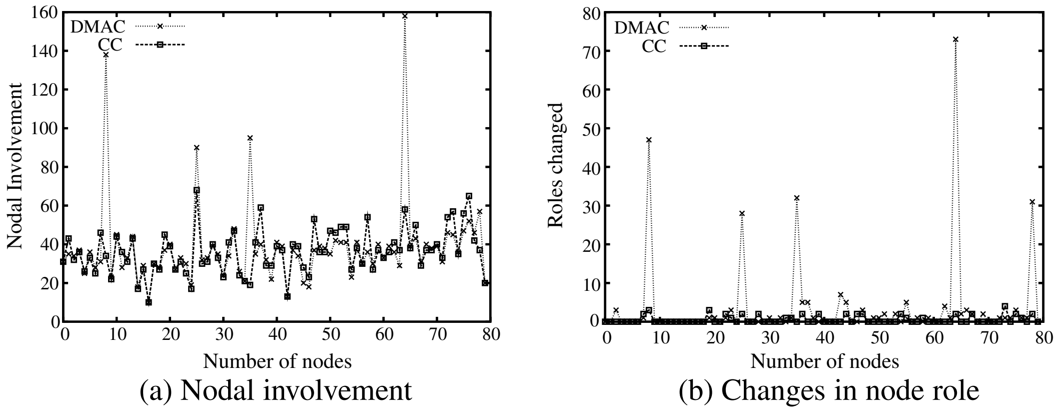


Fig. 16. Nodal involvement and changes in node role after a new node arrives in networks with 400 nodes.

after adding a node (Fig. 14(a)) is considerably less (up to a 62% less!) of the overhead imposed by DMAC. The overhead is primarily due to the messages used to inform the node about the new arrivals (e.g., the BEACON and NEW_NODE messages in CC), to the JOIN and INFO messages, as well as to the acknowledgments sent to confirm the reception of all the messages transmitted. In fact, the highest share of overhead is due to the acknowledgments. This is particularly true for broadcast messages, where many receiving nodes may answer the same broadcast message, each with an acknowledgment. For how small they maybe, the number of these acknowledgments may be quite high. After the acknowledgments, the second culprit of the overhead for nodal addition is the herd of JOIN messages. The difference in the amount of bytes transmitted to inform about an affiliation to a cluster (JOIN) in CC and DMAC shows again the core difference between the two approaches (Fig. 14(b)), and the validity of including mechanisms that are file-safe to nodal removal and addition into the design of backbone formation. In CC only a new node “changes” its role, becoming either a clusterhead or affiliating to one of the existing clusters. In DMAC the arrival of a new node can trigger a chain reaction which may lead (say, upon the arrival of a new node with high weight) to the reorganization of big parts of the network. This explains why the overhead of JOIN messages does not increase with N in CC while it “sky rockets” in DMAC. The difference between the overhead experienced by the two protocols for this type of message exchange can be as high as two order of magnitude when $N = 400$! Similar trends can be appreciated for the third major component of the total overhead: The INFO messages. Being the changes in role and backbone structure so limited in CC only routes to the a newly formed cluster or to the cluster the new node affiliated to have to be recomputed and communicated in CC. The more extensive changes imposed by DMAC demand for a much larger amount of INFO messages (Fig. 14(c)). The only phase of the backbone maintenance in which CC pays a higher overhead is when the nodes are informed about the presence of a new node, and exchange all the needed preliminary information for it to decide its role. In this phase the arriving node needs to contact a potentially large number of adjacent nodes, asking for additional information (such as the list of the nodes in their cluster) which are instead not needed in DMAC. This justifies why CC exchanges 2–3 times more bytes than DMAC. Overall, however, CC is much more efficient in limiting the exchange of information, triggering contained changes in the backbone and resulting in remarkably lower overhead.

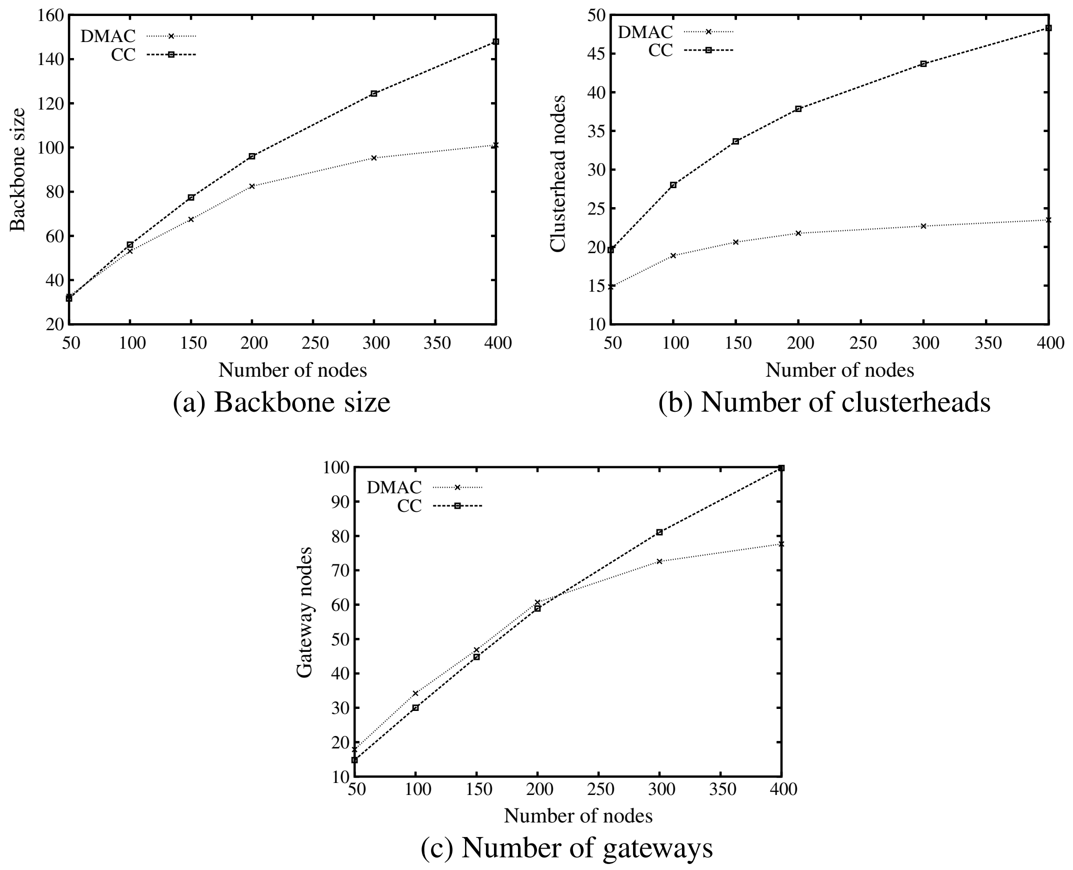


Fig. 17. Backbone size and node roles in the backbone.

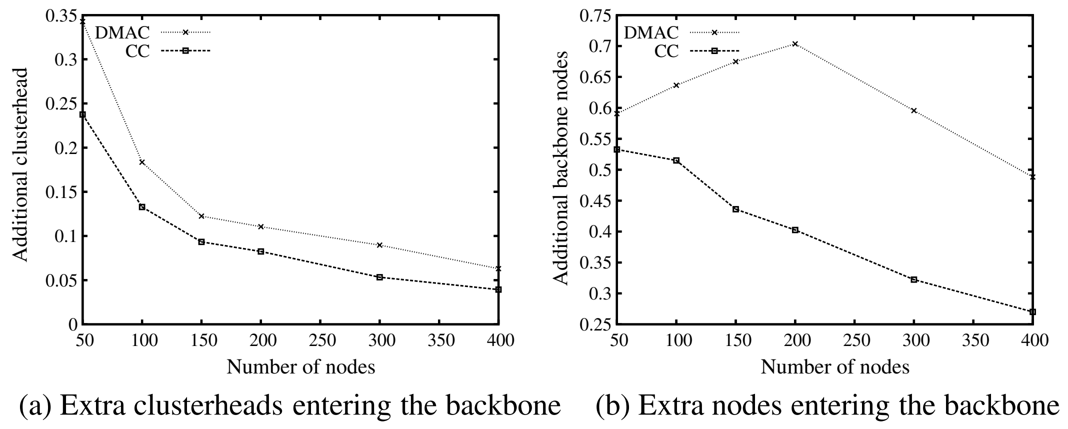


Fig. 18. Nodes entering the backbone after a new node arrives.

The lower number of messages and information to exchange, and the fact that this exchange happens locally, also explains why CC needs considerably less time for reorganizing the backbone after a node is added. This is clearly shown in Fig. 15 that depicts the average time needed to CC and DMAC to converge to a connected backbone after a node has been added (on average, and in the worst case). Both protocols behave quite well on average: Less than a couple of seconds are enough to reorganize the backbone. Things change considerably instead when we look at "worst case" scenarios. The maximum time for backbone maintenance is still around 2 s in CC. However, it can grow up to 20 s in DMAC!, as shown in Fig. 15(b).

To understand what happens in DMAC and how frequent the worst case happens, we have investigated the number of role changes (from ordinary to clusterhead, from clusterhead to ordinary, as well as the cases in which a non gateway node becomes gateway and viceversa) needed to deal with node additions over time (Fig. 16(b)). The *i*th point in the figure depicts the number of role changes caused by the addition of the *i*th node during the run of an experiment. (We have produced similar figures for all the performed experiments, and we have chosen to show a typical one). We notice that while the nodes changing role, entering or exiting the backbone as gateways, are just a few in CC, in DMAC some node additions (induced by the arrival of heavy nodes) can trigger an extremely high number of role changes in the backbone. Consider for example the case in which a node arrives which has the biggest weight among the current network nodes. This depicts quite a realistic case if the weight reflects the nodal residual energy: It is expected that new nodes will have fresh battery and hence much more energy with respect to the nodes that have already been in the network for some time. Being the heaviest in its neighborhood, the newly arrived node will become a clusterhead. This will change the role of several nodes which were previously affiliated to other clusters and that now will affiliate to the heavy, newly arrived node. Since the set of their members changed (some nodes have left for the new cluster) adjacent clusters will also have to reorganize, leading to new changes, and so on, in a chain of reactions. Moreover, during the wave-like propagation of the changes through the network, it may also happen that the same node changes its role several times before stabilizing to a correct, final one. The number of nodes involved in message exchange is displayed in Fig. 16(a), for the same run of our experiments. We observe that up to almost half of the DMAC nodes can be involved in a reorganization.

We now dig into the properties of the backbones resulting from a network reorganization after the addition of a node. The average backbone size, as well as the number of clusterheads and the number of gateways are depicted in Fig. 17.

Similarly to what seen for the first two groups of experiments, CC results in bigger backbones because of the smaller clusters (and thus of the higher number of clusterheads), and because of the need to interconnect such clusterheads to a higher number of adjacent ones. In CC the number of clusters is twice as many as the ones in DMAC. The number of gateway nodes is also higher in CC, even if the difference between the two protocols is only significant for very high densities ($N = 300$ and 400).

It is interesting to investigate how much the increase in backbone size depends on the originally built backbone or if it is instead due to the nodes that become part of the backbone after a new node is added. Figs. 18(a) and 18(b) depict the average number of clusterheads (and the total number of nodes) entering the backbone, on average, when a new node is added, for both CC and DMAC. The values are very small: In the majority of cases a new node can affiliate to the existing clusters with no change in the backbone. Being resilient to chain reactions CC is however the one showing better performance, even on average (and more significantly if we consider worst-case scenarios). A new node is more likely to affiliate to an existing clusterhead in CC, and the number of gateway nodes entering the backbone upon an addition is around half the nodes in DMAC.

6. Conclusions

We have described a new distributed and localized protocol for backbone formation in wireless sensor networks. *Clique clustering* (CC) partitions the network into cliques, thus providing each cluster with a fail-safe mechanism for quickly reconfiguring itself when nodes are added to or removed from the network. We have compared CC with DMAC, a previously proposed protocol for backbone formation and reorganization. Through thorough ns2-based simulation results we have investigated the properties of the clustering and backbones formed by the two algorithms. We have also explored the reactions of the two protocols to network dynamics, such as the removal and the addition of new nodes. We have clearly seen that CC is effective in reorganizing clusters and backbones very quickly (in always less that 4 s) and with remarkably lower overhead than DMAC (the latter having an overhead that is over 200% higher than CC's). Backbone formation and reorganization techniques render CC a solution for hierarchical organization of WSNs that efficiently deals with those network dynamics that are typical of WSNs.

References

[1] S. Basagni, M. Conti, S. Giordano, I. Stojmenovic (Eds.), Mobile Ad Hoc Networking, IEEE Press and John Wiley and Sons Inc., Piscataway, NJ and New York, NY, 2004.

[2] I.F. Akyildiz, W. Su, Y. Sanakarasubramaniam, E. Cayirci, Wireless sensor networks: A survey, Computer Networks 38 (4) (2002) 393–422.

[3] F. Zhao, L. Guibas, Wireless Sensor Networks. An Information Processing Approach. Networking, Morgan Kaufman, San Francisco, CA, 2004.

[4] A. Marcucci, M. Nati, C. Petrioli, A. Vitaletti. Hierarchical routing and data aggregation in wireless sensor networks. Technical report, Università di Roma "La Sapienza", Roma, Italy, 2003.

[5] S. Basagni, A. Carosi, C. Petrioli. Sensor-DMAC: Dynamic topology control for wireless sensor network, in: Proceedings of the 60th IEEE Vehicular Technology Conference, VTC 2004 Fall, September 26–29 2004, vol. 4, Los Angles, CA, 2004, pp. 2930–2935.

[6] S. Basagni. Distributed clustering for ad hoc networks, in: A.Y. Zomaya, D.F. Hsu, O. Ibarra, S. Origuchi, D. Nassimi, M. Palis (Eds.), Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99), June 23–25 1999, IEEE Computer Society, Perth/Fremantle, Australia, 1999, pp. 310–315.

[7] S. Basagni, M. Mastrogiovanni, A. Panconesi, C. Petrioli. Localized protocols for ad hoc clustering and backbone formation: a performance comparison, in: S. Olariu, D. Simplot-Ryl, I. Stojmenovic (Eds.), IEEE Transactions on Parallel and Distributed Systems, Special Issue on Localized Communication and Topology Protocols for Ad Hoc Networks 17(4) (2006) 292–306.

[8] S. Park, K. Shin, A. Abraham, S. Han, Optimized self organized sensor networks, Sensors 7 (2007) 730–742.

[9] C. Johnen, L.H. Nguyen, Self-stabilizing weight-based clustering algorithm for ad hoc sensor networks, in: Proceedings of the Second International Workshop on Algorithmic Aspects of Wireless Sensor Networks, ALGOSENSOR 2006, June 15 2006, Venice, Italy, 2006, pp. 83–94.

[10] K. Lu, J.A. Bolla, D.T. Huynh, Adapting connected D-hop dominating sets to topology changes in wireless ad hoc networks, in: Proceedings of the 25th IEEE International Performance, Computing

and Communications Conference, IPCCC 2006, April 10–12 2006, Phoenix, AZ, 2006, pp. 207–214.

[11] P. Krishna, N.H. Vaidya, M. Chatterjee, D.K. Pradhan, A cluster-based approach for routing in dynamic networks, ACM SIGCOMM Computer Communication Review 27 (2) (1997) 49–64.

[12] Y. Fernandess, D. Malkhi, *k*-clustering in wireless ad hoc networks, in: Proceedings of the Second ACM International Workshop on Principles of Mobile Computing, POMC 2002, October 30–31 2002, Toulouse, France, 2002, pp. 31–37.

[13] P. Tosic, G. Agha, Maximal clique-based distributed group formation for autonomous agent coalitions, in: Third International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2004, August 19–23 2004, IEEE Computer Society, New York, NY, USA, 2004.

[14] R. Gupta, J. Walrand, Approximating maximal cliques in ad-hoc networks, in: Proceedings of PIMRC 2004, September 5–8 2004, vol. 1, Barcelona, Spain, 2004, pp. 365–369; Proceedings of the 15th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2004.

[15] K. Sun, P. Peng, P. Ning, C. Wang. Secure distributed cluster formation in wireless sensor networks, in: Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC 2006, Washington, DC, USA, IEEE Computer Society, 2006, pp. 131–140.

[16] B.N. Clark, C.J. Colbourn, D.S. Johnson, Unit disk graphs, Discrete Mathematics 86 (1990) 165–167.

[17] R. Gupta, J. Walrand, O. Goldschmidt, Maximal cliques in unit disk graphs: polynomial approximation, in: Proceedings of INOC 2005, March 2005, Lisbon, Portugal, International Network Optimization Conference, 2005.

[18] I. Chlamtac, A. Faragó, A new approach to the design and analysis of peer-to-peer mobile networks, Wireless Networks 5 (3) (1999) 149–156.

[19] S. Basagni, C. Petrioli, R. Petroccia, Clique clustering. Technical Report 09/2006, Dipartimento di Informatica, Università di Roma "La Sapienza", Roma, Italy, August 2006.

[20] S. Basagni, Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks, in: Proceedings of the IEEE 50th International Vehicular Technology Conference, VTC 1999-Fall, September 19–22 1999, vol. 2, Amsterdam, The Netherlands, 1999, pp. 889–893.

[21] S. Basagni, M. Elia, R. Ghosh, ViBES: Virtual backbone for energy saving in wireless sensor networks, in: Proceedings of the IEEE Military Communication Conference, MILCOM 2004, October 31–November 3 2004, Monterey, CA, 2004.

[22] The VINT Project, The ns Manual, <http://www.isi.edu/nsnam/ns/>, 2002.

[23] P.J.M. Havinga, S. Etalle, H. Karl, C. Petrioli, M. Zorzi, H Kip, T. Lentsch. EYES– energy efficient sensor networks, in: M. Conti, S. Giordano, E. Gregori S. Olariu (Eds.), Proceedings of the Fourth IFIP TC6/WG6.8 International Conference on Personal Wireless Communications (PWC) LNCS 2775, September 23–25 2003, Venezia, Italy, 2003, pp. 198–201.

Stefano Basagni holds a Ph.D. in electrical engineering from the University of Texas at Dallas (December 2001) and a Ph.D. in computer science from the University of Milano, Italy (May 1998). He received his B.Sc. degree in computer science from the University of Pisa, Italy, in 1991. Since Winter 2002 he is on faculty at the Department of Electrical and Computer Engineering at Northeastern University, in Boston, MA. From August 2000 to January 2002 he was professor of computer science at the Department of Computer Science of the Erik Jonsson School of Engineering and Computer Science, The University of Texas at Dallas.

Dr. Basagni's current research interests concern research and implementation aspects of mobile networks and wireless communications systems, Bluetooth and sensor networking, definition and performance evaluation of network protocols and theoretical and practical aspects of distributed algorithms.

Dr. Basagni has published over five dozens of refereed technical papers and book chapters. He is also co-editor of two books. Dr. Basagni served as a guest editor of the special issue of the Journal on Special Topics in Mobile Networking and Applications (MONET) on Multipoint Communication in Wireless Mobile Networks, of the special issue on mobile ad hoc networks of the Wiley Interscience Wireless Communications & Mobile Networks Journal, of the special issue of the Elsevier's journal Ad Hoc Networks on advances in ad hoc and sensor networking, and of the special issue of the Elsevier's journal Algorithmica on algorithmic aspects of mobile computing and communications.

Dr. Basagni serves as a member of the editorial board and of the technical program committee of ACM and IEEE journals and International conferences. He is a senior member of the ACM (including the ACM SIGMOBILE), senior member of the IEEE (Computer and Communication societies), and member of ASEE (American Society for Engineering Education).

Chiara Petrioli received the Laurea degree "summa cum laude" in computer science in 1993, and the Ph.D. degree in computer engineering in 1998, both from Rome University "La Sapienza", Italy. She is currently Associate Professor with the Computer Science Department at Rome University "La Sapienza", Her current work focuses on Ad hoc and sensor networks, Delay Tolerant Networks, Personal Area Networks, Energy-conserving protocols, QoS in IP networks and Content Delivery Networks where she contributed around sixty papers published in prominent International journals and conferences. Prior to Rome University she was research associate at Politecnico di Milano and was working with the Italian Space agency (ASI) and Alenia Spazio. Dr. Petrioli was guest editor of the special issue on "Energy-conserving protocols in wireless Networks" of the ACM/Kluwer Journal on Special Topics in Mobile Networking and Applications (ACM MONET) and is associate editor of IEEE Transactions on Mobile Computing, the ACM/Kluwer Wireless Networks journal, the Wiley InterScience Wireless Communications & Mobile Computing Journal and the Elsevier Ad Hoc Networks journal. She has served in the organizing committee and technical program committee of several leading conferences in the area of networking and mobile computing including ACM MobiCom, ACM MobiHoc, IEEE ICC, IEEE Globecom, IEEE INFOCOM. She is member of the steering committee of ACM Sensys and of the International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous) and serves as member of the ACM SIGMOBILE executive committee. Dr. Petrioli was a Fulbright scholar. She is a senior member of IEEE and a member of ACM.

Roberto Petroccia received the Laurea degree "summa cum laude" in computer science in 2006 from Rome University "La Sapienza", Italy. He is currently a Ph.D. student at the Computer Science Department at Rome University "La Sapienza". His current work focuses on ad hoc and sensor networks, delay tolerant networks and energy-conserving protocols.