社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－AL－106 (5)
2006／5／18

動的なセンサー網に対するクラスタに基づいたアーキテクチャの比較

内田 次郎† Muzahidul A.K.M Islam† 稲葉 直貴†
片山 喜章† 陳 慰†† 和田 幸一†

† 名古屋工業大学
†† テネシー州立大学

あらまし センシング機能，通信機能を備えた小型センサーデバイスから構成されるネットワークをセンサーネットワークという．センサーネットワークに対し，"オーバーヘッドやエネルギー消費量の最小化"などの利点を持つアーキテクチャの構築方法としてクラスタリングが挙げられる．本研究では，[2] で提案されているアーキテクチャが持つ望ましい性質を維持しつつ改良を加え，タスクの完了時間やその拡張などの面においてよりよい性質を持った三つのアーキテクチャおよびそのメンテナンスのためのアルゴリズムを提案する．

Toward better cluster-based architectures for Ad hoc sensor networks

Jiro UCHIDA†, Muzahidul A.K.M ISLAM†, Naoki INABA†, Yoshiaki KATAYAMA†,
Wei CHEN††, and Koichi WADA†

† Nagoya Institute of Technology
†† Tennessee State University

Abstract A sensor network is a collection of transmitter-receiver devices (referred to as nodes). Clustering is seen as the step to provide the flat sensor network topology with a hierarchical architecture with properties such as minimizing communication overhead and minimizing the overall power consumption. In this paper we improve the architecture [2] maintaining the desirable properties and propose three architecture and maintenance algorithms which has the better properties about the completion time of tasks, expansions of tasks, and so on.

1. Introduction

A wireless network is a collection of transmitter-receiver devices (denoted as nodes) which can communicate with each other via radio.

There is increasing interest in self-organizing multihop wireless networks composed of a large number of autonomous nodes communicating via radio without any additional infrastructure. These nodes can be static or mobile, and they are usually constrained as for the critical resources, such as power and radio frequency band. A typical example is given by wireless sensor networks (WSNs), where sensor nodes are usually irreplaceable, and become unusable after energy depletion or other failures.

After the nodes of a sensor network are deployed physically, a *flat* network topology is formed in which a link exists between two nodes if they are in each others communication range. In such a flat network topology, there is no established structure on which the nodes could take efficient communication.

Clustering is seen as the step to provide the flat sensor network topology with a *hierarchical* organization. The basic idea is that of breaking the network into physical proximity clusters which are smaller in scale and usually simpler to manage by the nodes called as *cluster head.* The subsequent *backbone* construction uses the clustering induced hierarchy to form a structure which mainly con-

sists of cluster heads and provides the communication between the clusters. The structured network is functional in providing desirable properties such as minimizing communication overhead, choosing data aggregation points, increasing the probability of aggregating redundant data, and minimizing the overall power consumption [1].

Considering the mobility and scalability, we need the operations such as *nodes getting out* of and *nodes joining into* an existing network. Even for stationary nodes, when battery is low, it must get out and go to charge mode. Then, the charged nodes should join back to the network once again. Therefore, once a hierarchical clustering established, the maintenance of the cluster organization turns to be crucial in the presence of network topology changing.

There are many discussion about distributed clustering of a flat network. Although many efforts have been made for establishment of a hierarchical clustering on a network, and the research for the maintenance of the cluster organization under the similar scenario is seldom seen, [2] puts emphasis on the maintenance of the clustering structure of the sensor network.

In [2], they consider a sensor network in which the network topology dynamically changes, and proposed a novel cluster structure on which two operations *node-move-in* and *node-move-out* are defined for maintaining the cluster organization. This work is based on the following radio network model [4]: each node has a distinct ID, nodes transmit or receive message in each round, and all nodes use a single radio channel without collision detection capability. Nodes in a flat network G are grouped into disjoint clusters, and a backbone consist of cluster heads (simply called heads) and gateway nodes (simply called gateways). Gateways are used in order to connect heads. A spanning tree can be obtained from a backbone and clusters, and the tree is called a *cluster-based network* $CNet(G)$. Let n be the number of nodes in G and p be the number of clusters, $CNet(G)$ has some following new properties:

(1) A backbone consists of at most $2p-1$ nodes;

(2) p is at most a cardinality of the minimum clique partition, $p \ll n$ in a dense graph;

(3) Broadcast on G can be performed via a backbone in $O(p)$ rounds (on a flat network, it requires at least $\Omega(n)$ rounds [5]);

(4) If G is unit disk graph, $p \leqq 5 \times |MDS|$.

In this paper we improve the architecture (called $\mathcal{C}$) [2] maintaining the desirable properties, and propose three architecture and maintenance algorithms which has the better properties about the size of a backbone, expansions of tasks, and so on. We call the first architecture as $\mathcal{M}$. In $\mathcal{C}$, only join/leave for one existing cluster-based network is dealed, *merge* of two or more cluster-based networks caused by join of a node and *separation* of cluster-based networks are not considered. Even if a merge is done, there exist the case which requires linear time for the number of nodes in $\mathcal{C}$. But in $\mathcal{M}$, a merge (and separation) can be performed more faster. In the second architecture $\mathcal{I}$, by extinguishing gateway, join/leave operation can be performed more flexible and the completion time is improved as well as $\mathcal{M}$. By changing $\mathcal{I}$ slightly, in the last architecture $\mathcal{U}$, the size of $BT(G)$ is decreased and it is verified that broadcast can be performed more fast by simulation.

In our clustering, when a $CNet(G)$ is established, each node has only one hop knowledge (i.e., each node knows their neighbors in the backbone, $CNet(G)$ and G, respectively). We will show that a $CNet(G)$ can be established either in a static way which means that all topological information are gathered somewhere and the problem is solved there, or in a dynamic way which means that each node solves the problem locally without gathering all information, in $O(n)$ time or in $O(|E|)$ time, respectively. The operations join and leave maintain the cluster structure for G with the same properties when a node gets out of or joins into G. Table 1. summarize our proposed maintenance algorithms for these operations and architectures with the properties maintained by it.

In this paper, we will omit the proofs of the lemmas and theorems, which can be found in our technical report [7].

2. Model and Definition

2.1 Model of Sensor Networks

The model of a sensor network G in this paper is as follows:

- Nodes repeat transmissions and receptions in fixed intervals, called *rounds*. In each round, each node acts as either a transmitter or a receiver.
- A round of for node is synchronized (i.e., he start and the end of each round is the same for each node).
- A node acting as a receiver in a given round gets a message iff exactly one of its neighbors transmits in this round. When more than one neighbor transmits simultaneously in a given round, collision occurs and none of the messages is received in this round.
- A node can not notice the occurrence of a collision, i.e., there is no collision detection in the network.
- Each node knows its ID, which is distinct for every node.

3. A Cluster-based Architecture

In this section, first we show the existing architecture [2] and basic structure that is common to our three proposed architectures, then a broadcast algorithm for such structure is shown. Next, we present the structure of each proposed architecture and its maintenance algorithm for it.

3.1 Common Definitions

Let $G = (V, E)$ be a connected bi-directional graph. Each node has a role called *status*. Nodes that comprise a dominating set of G are called *head*, and a subgraph of G which satisfies following conditions and consists of one head and other nodes except head is called *cluster*: Nodes except head is called *member*, and every member is adjacent to a head in each cluster and there is no edge between members. A cluster is a star subgraph of G with a head as a center. Each head forms one cluster and no node belongs to two or more clusters.

A backbone tree $BT(G)$ of G is a subtree of G including all heads. As G is a connected bi-directional graph, a backbone tree must exist. All nodes that are not in $BT(G)$ belong to some cluster as member.

Backbone tree can be considered as a communication highway on G. To see this, let u and v be two members, and h_u and h_v be their heads, respectively. If u wants to

Table. 1 Properties of our architectures

		$\mathcal{C}$	$\mathcal{M}$	$\mathcal{I}$	$\mathcal{U}$
$\|BT(G)\|$	a case without merge	$\leqq 2p_G - 1$	$\leqq 2p_G - 1$	$\leqq 2p_G - 1$	$\leqq 2p_G - 1$
	a case with merge	-	$\leqq 3p_G - 2$	$\leqq 2p_G$	$\leqq 2p_G$
the number of cluters		$\leqq p_G$	$\leqq p_G$	$\|BT(G)\|$	$\|BT(G)\|$
status		head gateway member	head gateway member	head member	head member1,2
Completion time for merge of *CNets*		-	$O(q + \max\{\|BT(G)\| \text{ for each } CNet\})$		
Completion time for separation of *CNets*		$O(\|T\|)$			

p_G : cardinality of the minimum clique partition,
n : the number of nodes in G,
q : the number of neighbors of joining node in G,
T : subtree of $CNet(G)$ with leaving node as root.

send message to v, u first sends the message to its head h_u. And h_u sends the message to h_v via $BT(G)$, then h_v sends the message to its member v.

Now we use the backbone tree to connect the clusters for forming a structured network on G. *A cluster-based network* of $G = (V, E)$ is a rooted tree $CNet(G) = (V, E_{BT(G)} \cup E_C)$ with one cluster head as a root, where the edges of $E_{BT(G)}$ come from the backbone tree and the edges of E_C come from all the clusters (see Fig 1). $CNet(G)$ forms a spanning tree of G.

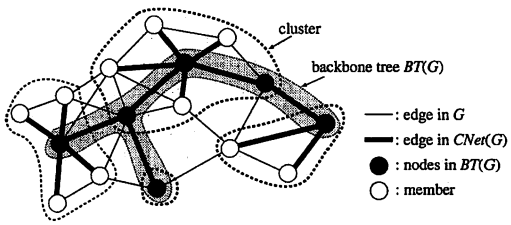

Fig. 1 G, $BT(G)$, $CNet(G)$

In the following sections, we will show how a flat graph G can self-organize and self-maintain itself into a cluster-based network $CNet(G)$. Before we discuss the algorithms, we first declare the data structure for $CNet(G)$ clearly.

A $CNet(G)$ has two level structures: a set of clusters, and a backbone tree which is used to connect the clusters. Each node v in G maintains the information described below:

- $v.stat$: v's status.
- $v.prt, v.chd$: an ID of v's parent and a set of IDs of v's children on $CNet(G)$, respectively. For a root r, $r.prt = \perp$, and for each node m who has no child, $m.chd = \emptyset$.
- $v.oneigh$: a set of IDs of v's neighbors on G except $v.prt$ and $v.chd$.
- $v.rootID$: an ID of a root of $CNet(G)$ to which v belongs.

Each node maintains its neighbor's status and ID as a pair.

Hereafter we use $v.neigh$ as the neighbors of v on G and $v.bneigh$ as the neighbors of v on $BT(G)$, respectively (these are derived from $v.prt$, $v.chd$ and $v.oneigh$).

We call above information as *total 1-hop data*. When the information are maintained for each node v in G, it is called that G *is organized* with total 1-hop data.

We define two operations *join* and *leave* on a $CNet(G)$.

- join: a new node v announces itself by sending a message to join the existing $CNet(G)$ and the network re-organizes itself to a new $CNet(G')$, where G' is the graph obtained by adding v to G.
- leave: a node v of $CNet(G)$ announces itself by sending a message to leave the existing $CNet(G)$, and the network re-organizes itself to a new $CNet(G')$, where G' is the graph obtained by removing v from G.

We call following operation as *merge*: Given disjoint graphs $G_1, G_2, \ldots, G_i$ and $CNet(G_1)$, $CNet(G_2)$, ..., $CNet(G_i)$, these graphs are connected by join of a new node and re-organized to one $CNet(G)$. It is considered as a part of join. We also call *separation* as follows: Given graphs G and $CNet(G)$, G is divided into disjoint graphs $G_1, G_2, \ldots, G_j$ by leave of a node and these graphs are re-organized to $CNet(G_1)$, $CNet(G_2)$, ..., $CNet(G_j)$. It is considered as a part of leave.

3.2 Broadcast

In this section, we show the broadcasting algorithm [2] using $CNet(G)$. In the algorithm, a broadcasting technique shown in [4] is used, and we call it as procedure *Eulerian*.

Eulerian(H) performs a broadcast on a bi-directional graph H. A message called token starts from the source node, visits every node and turns to the source node. At the beginning, the token is in the source node. It then visits each node in H from the source node in depth-first order. When node v gets the token, it sends the token with the message and its ID to one of its neighbors which have not received the token yet. If v has no neighbor which has not been visited by the token yet, v returns the token to the node from which it got the token for the first time. The movement of the token forms an Eulerian cycle of H. It patrols every node in H and returns to the source node finally.

The following lemma holds for *Eulerian*.

Lemma 1. [4] *Let $H = (V, E)$ be a connected bi-directional graph. If each node of H knows all its neighbors' IDs in H, procedure Eulerian(H) completes broad-*

casting for H in $O(|V|)$ rounds.

We now show the broadcasting algorithm BroadcastALG [2] in $CNet(G)$, where s is the source node with a message M and needs to be informed to the rest of the nodes in a given network G.

Since every node in a graph on which *Eulerian* is performed transmits certainly and only one node transmits in each round, and $BT(G)$ is consist of a dominating set of G, a broadcasting in $CNet(G)$ can be completed by performing *Eulerian* on $BT(G)$. Algorithm 1 shows the algorithm BroadcastALG.

Algorithm 1 BroadcastALG

```
1: if a source node s is not in BT(G) then
2:     s sends source message M to s.prt;
3:     s.prt calls procedure Eulerian(BT(G));
4: else
5:     s calls procedure Eulerian(BT(G));
6: end if
```

Theorem 1. [2] *Let $CNet(G)$ be a cluster-based network of G and $BT(G)$ be a backbone tree $BT(G) = (V_{BT}, E_{BT})$ of G. A broadcasting on $CNet(G)$ can be done in $O(|V_{BT}|)$ rounds.*

3.3 $\mathcal{C}$

We show the architecture $\mathcal{C}$ [2]. In addition to the definition in subsection 3.1, on $\mathcal{C}$ a set of all heads is a independent set of G. Nodes in $BT(G)$ except for heads are called *gateway*. On $BT(G)$, gateways do not adjacent to each other and adjacent to two or more heads.

Refer to [2] about join/leave algorithms that maintain such structure $\mathcal{C}$.

$\mathcal{C}$ has the following properties.

Lemma 2. *Let G be a connected bidirectional graph and $BT(G)$ be a backbone tree of G. If $BT(G)$ has p heads, then the number of nodes in $BT(G)$ is at most $2p-1$ nodes on $\mathcal{C}$.*

Following lemmas hold by the fact that heads are not adjacent to each other on G.

Lemma 3. *Let G be a connected bidirectional graph and p_G be the cardinality of minimum clique partition of G. The number of heads in $CNet(G)$ is at most p_G.*

Lemma 4. *Let $G = (V, E)$ be a unit disk graph, and $MDS(G)$ be the minimum dominating set of G. The number of heads in a $CNet(G)$ is not larger than $5 \times |MDS(G)|$.*

In [2], a merge operation is not considered.

3.4 $\mathcal{M}$

On architecture $\mathcal{C}$, a merge of two or more cluster-based networks caused by a join of a node is not considered.

So a new architecture $\mathcal{M}$ is proposed in [3] that enable a merge by allow gateways to adjacent to each other on $BT(G)$.

As well as $\mathcal{C}$, a set of all heads is a independent set of G on $\mathcal{M}$. All of leaves on $BT(G)$ are heads, and nodes in $BT(G)$ except for heads are called gateways. Each gateway is adjacent to at least one head on $BT(G)$.

Refer to [2] about join/leave algorithms that maintain such structure $\mathcal{M}$.

$\mathcal{M}$ has the following property.

Lemma 5. *Let G be a connected bidirectional graph and $BT(G)$ be a backbone tree of G. If $BT(G)$ has p heads, then the number of nodes in $BT(G)$ is at most $3p-2$ nodes on $\mathcal{M}$.*

By the fact that heads are not adjacent to each other on G, lemmas 3, 4 are also hold as well as $\mathcal{C}$.

On $\mathcal{M}$, a merge can be performed without changing the order of the size of a backbone tree, i.e. the order of completion time for broadcast are not changed. And in the case that a merge is not caused, the size of a backbone tree and the completion times of join/leave algorithms are the same as [2].

3.5 $\mathcal{I}$

3.5.1 Structure

On architecture $\mathcal{M}$, although a merge can be performed without increasing the order of completion time for each operation, status *gateway* is used except for head and member, and the condition of backbone tree is also complicated. So we propose the architecture $\mathcal{I}$ constructed by only heads and members.

On $\mathcal{I}$, a set of heads is not independent set of G and a backbone tree is constructed by only all heads.

Let $CNet_{\mathcal{I}}(G)$ and $BT_{\mathcal{I}}(G)$ be $CNet(G)$ and $BT(G)$ with architecture $\mathcal{I}$, respectively.

3.5.2 Maintenance Algorithms

First we show the join algorithm for $\mathcal{I}$.

Let *new* be a node who wants to join a network $G = (V, E)$, wherer G consists of disjoint subgraphs $G_1, G_2, \ldots, G_i$. And let $G' = (V \cup \{new\}, E \cup E_{new})$ in this subsection, where $E_{new} = \{(u, new) | u$ is in transmitting range of the node *new*, $u \in V\}$. Let $q = |E_{new}|$. We simply use "neighbors" as "neighbors in G'" in this subsection.

What should be performed by join operation is to decide the status of *new* and update the information which the neighbors of *new* in G' have.

First to decide the status of *new* and whether a merge is caused, *new* needs to know the status of its neighbors and their rootIDs. In case that *new* receives only one rootID, no merge is caused. Then, if there exist heads in the neighbors of *new* in G', *new* selects one to be it's head and itself becomes a member. Else there are no heads in its neighbors, *new* becomes a head and sets one neighboring member to be the head. Based on the status of *new* has decided, the neighbors of *new* update their information. In no matter which cases, the process affects only 2-hop neighbors of *new* in G'.

In case that *new* receives two or more rootIDs, a merge is caused. Let R be the graph consist of a cluster-based network with rootID r and *new*. In a merge, first *new* determines the rootID. Then, nodes in $BT_{\mathcal{I}}$ for each $CNet_{\mathcal{I}}$ change their status preferentially by a join without merge, and move into R one by one. R grows as nodes move into it, and we also denote the grown graph as R. When they become head, since they consist of dominating sets, member nodes from the first in each $CNet_{\mathcal{I}}$ can be children of them. Otherwise, when a node b does not become head, check neighboring members of b whether they are adjacent to other heads in R. If heads in R is

adjacent to them, they set their parent to the head. Then if all member become children of other heads, b remains member, else b and one of the members become head.

In order to determine the status of *new* and update the information, it is sufficient that the neighbors of *new* transmit their own IDs and status one by one. It can be done by numbering the neighbors of *new* from 1 to q and transmitting their information in order of the numbers.

Lemma 6. [2] *The neighbors of new can be numbered from 1 to q in $O(q)$ expected rounds, where q is the number of neighbors of new in G'.*

Now, we present our join algorithm $\mathcal{I}$-join and subroutines of it in Algorithm 2, Procedure 3, 4, and 5 to maintain the architecture $\mathcal{I}$, respectively.

Algorithm 2 $\mathcal{I}$-join

```
1: The joining node new sends AddMe message;
2: The nodes receiving AddMe message are numbered from
   1 to q, and send their IDs and status to new one by one;
3: if new does not receives two or more rootIDs then
4:    I-status;
5: else
6:    I-merge;
7: end if
```

Procedure 3 $\mathcal{I}$-status

```
1: if there are heads in neighbors of new then
2:    new sends I'mMember message to the neighboring head
      h with minimum ID;
3:    new.rootID := h.rootID, new.stat := member,
      new.prt := h, h.chd := h.chd ∪ {new};
4: else {only member}
5:    new sends BeHead message to a neighboring member
      m with minimum ID;
6:    new.rootID := h.rootID,
      new.stat := head, m.stat := head,
      new.prt := m, m.chd := m.chd ∪ {new};
7:    m sends ChgHead message to its neighbors;
8:    Neighbors of m change m'status into head in their in-
      formation;
9: end if
```

Lemma 7. *Let $CNet_{\mathcal{I}}(G)$ be a cluster-based network of G. When G is organized with total 1-hop data, after an execution of $\mathcal{I}$-join for a node new, G' is organized with total 1-hop data.*

Theorem 2. *For disjoint graphs G_1, G_2, ..., G_i and $CNet_{\mathcal{I}}(G_1)$, $CNet_{\mathcal{I}}(G_2)$, ..., $CNet_{\mathcal{I}}(G_i)$, when these graphs are organized with total 1-hop data, join of new can be done in $O(q)$ expected rounds with no merge or in $O(q + \max\{|BT(G)| \text{ for each } CNet\})$ expected rounds with merge, and G' is organized with total 1-hop data, where q is the number of neighbors of new in G'.*

In the execution of $\mathcal{I}$-join only the neighbors of *new*, and neighbors of the neighbors of *new* receive a message caused by join. Hence, join can be performed locally in only 2-hop neighbors of *new* without changing status of other nodes.

Procedure 4 $\mathcal{I}$-merge

```
1: % Let r be the minimum rootID in neighbors of new;
2: % Let R = G[{v|v ∈ CNet_I with rootID r}∪{new}∪{v|v
   has called I-status}];
3: % Let t be a node with a token in Eulerian;
4: new call I-status for CNet_I with rootID r;
5: for each rootID i of new's neighbors do
6:    if there is no head in new's neighbors with rootID i
      then
7:       new sends DoJoin message to a neighboring member
         u with rootID i and minimum ID;
8:    else
9:       new sends DoJoin message to a neighboring head u
         with rootID i and minimum ID;
10:   end if
11:   u call I-status for R;
12:   u calls Eulerian on the backbone with rootID i which
      works as follows in each round:
13:      t joins into R by I-status;
14:      if t becomes member then
15:         deliver-member(t);
16:      end if
17:      Nodes in t.chd set their parent into t;
18: end for
```

Procedure 5 deliver-member(v)

```
1: v.mlist := v.chd − v.bneigh;
2: v.chd := ∅;
3: v.stat := member;
4: while v.mlist ≠ ∅ and v.stat = member do
5:    v sends ChkM message to a member m ∈ v.mlist;
6:    if there is no head in m.neigh then
7:       m calls I-status as v is its parent for R;
8:    else
9:       m sends GetNM message to one head h ∈ m.neigh;
10:      h sends its neighboring members' IDs NM to v via
         m;
11:      v sends CoveredM message with CM := v.mlist ∩
         NM, and nodes in CM set their parent to h;
12:      m sends CM to h, and h.chd := h.chd ∪ CM;
13:      v.mlist := v.mlist − CM;
14:   end if
15: end while
16: v.chd := v.mlist;
17: Nodes in v.chd set their parent into v;
```

Next, we show our leave algorithm $\mathcal{I}$-leave to maintain $\mathcal{I}$.

Let *lev* be a node who wants to leave from G and G' be the graph after *lev* leaves, that is, $G' = G[V - \{lev\}]$ in this subsection. It is possible to judge whether G' is connected in $O(|T|)$ rounds, where $T = (V(T), E(T))$ is a subtree of $CNet_{\mathcal{I}}(G)$ with the leaving node *lev* as the root (see Fig.2).

Our leave algorithm is executed when *lev* wishes to leave from the network. If *lev* is a member, it sends *I'mLeaving* message and simply leaves from the network. Otherwise, the leave algorithm works as follows: First, we consider the case where *lev* is not the root of $CNet_{\mathcal{I}}(G)$. The case where *lev* is the root is described later. If *lev* is a head, $CNet_{\mathcal{I}}(G)$ is divided into two subtrees. One is the tree T with *lev* as the root (not including the root

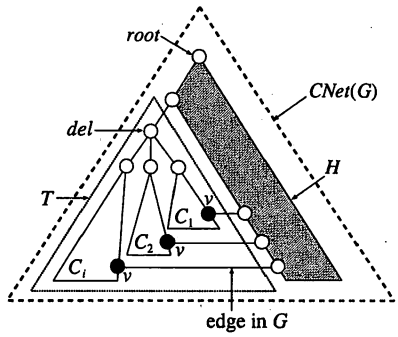

Fig. 2 $CNet_{\mathcal{I}}(G)$, subtree T, connected components of T

in $CNet_{\mathcal{I}}(G)$), and one is the tree H with the root of $CNet_{\mathcal{I}}(G)$ as the root. The algorithm $\mathcal{I}$-leave removes *lev* from T, and adds other nodes of T to H so that the resulted tree is $CNet_{\mathcal{I}}(G)$.

Let $C_i(i = 1, 2, \ldots)$ be the connected components of $G[V(T) - \{lev\}]$ (Fig.2). H always changes and grows larger each time when a node in T is added to H.

The edges in G between T and H are used in order to add the nodes of T to H. By using join operation described above, the nodes in T can be merged into H. Each node already knows its neighbors in G and their status in $CNet_{\mathcal{I}}(G)$, therefore, a join operation can be performed deterministically in $O(1)$ rounds. First, *lev* calls *Eulerian*(T) to wake up each node of T. Whenever the waken node $v \in C_j$ has an edge connected with H, v moves to H by $\mathcal{I}$-join, then v calls *Eulerian*(C_j) and each node in C_j moves to H following v by $\mathcal{I}$-join one by one. The above process will be repeated until all of *lev*'s children have received a token or moved to H. If all of the nodes in T other than *lev* are moved to H, the leave operation is completed. Else the remaining nodes devided into connected components, then separation is performed and *CNets* of these components are constructed.

Here we describe about an exception, when *lev* is a root of $CNet_{\mathcal{I}}(G)$. If *lev.bneigh* $\neq \emptyset$, electing a head which is 2-hop neighbor of *lev* and setting it to a new root of $CNet_{\mathcal{I}}(G)$, our algorithm in the general case can be used. Otherwise *lev* is the only head in $CNet_{\mathcal{I}}(G)$, select one node in *lev.neigh* becomes the new root of $CNet_{\mathcal{I}}(G')$. The new root calls *Eulerian*$(G[V - \{lev\}])$ and a cluster-based network is constructed sequentially by repeating join for the node with token.

Our leave algorithm $\mathcal{I}$-leave is described in Algorithm 6 and Procedure 7.

Lemma 8. *Let $CNet_{\mathcal{I}}(G)$ be a cluster-based network of G. When G is organized with total 1-hop data, after an execution of $\mathcal{I}$-leave for a node lev, G' is organized with total 1-hop data.*

Theorem 3. *Let $CNet_{\mathcal{I}}(G)$ be a cluster-based network of G and T be the subtree of $CNet_{\mathcal{I}}(G)$ with the leaving node lev as root. When G is organized with total 1-hop data, leave of lev can be done in $O(|T|)$ rounds, and G' is organized with total 1-hop data.*

3.5.3 Properties

$\mathcal{I}$ which is constructed above two operations has the following property.

Lemma 9. *The number of clusters is equal to $|BT_{\mathcal{I}}(G)|$.*

Lemma 10. *Let G be a connected bidirectional graph and p_G be the cardinality of minimum clique partition of G. $|BT_{\mathcal{I}}(G)|$ is at most $2p_G$ on $\mathcal{I}$ which is constructed*

Algorithm 6 $\mathcal{I}$-leave

% Let $T = (V(T), E(T))$ be a subtree of $CNet(G)$ with root *lev*;
% Let $C_i(i = 1, 2, \ldots)$ be the connected components of $G[V(T) - \{lev\}]$;
% Let $H = G[V - V(T) \cup \{v | v \in V(T), v$ has called $\mathcal{I}$-join$\}]$;
% Let t be a node with a token in *Eulerian*;

1: **for** each $v \in G$ **do**
2: $v.link := v.neigh$;
3: **end for**
4: *lev* sends *I'mLeaving* message;
5: **if** *lev.stat* = member **then**
6: nodes that received *I'mLeaving* delete *lev* from neighbor list in G;
7: **else**
8: **if** *lev* is a root of $CNet(G)$ **then**
9: **if** *lev.bneigh* $\neq \emptyset$ **then**
10: *lev* sends *chkchild* message;
11: Each node $v \in$ *lev.bneigh* sends *v.child* to *lev* one by one;
12: **end if**
13: **if** There is a head 2-hop away from *lev* **then**
14: *change-root*;
15: **else**
16: *exception*; **exit**;
17: **end if**
18: **end if**
19: **if** *lev* is a head and $|(lev.prt).bneigh| = 2$ **then**
20: $(lev.prt).chd := (lev.prt).chd - lev$;
21: *deliver-member(lev.prt)*;
22: Nodes in $(lev.prt).chd$ set their parent into *lev.prt*;
23: **end if**
24: % Let $T' := T$;
25: *lev* calls *Eulerian*(T) which works as follows in each round:
26: $v.link := v.link - \{t\}$ for each node v who receives messages from t;
27: **while** there is a node in *lev.chd* who has not received a token and not joined to H **do**
28: *lev* calls *Eulerian*(T'), and it works as follows in each round:
29: **if** $t.link \neq \emptyset$ **then** the procedure finishes;
30: $t \in C_j$ calls *Eulerian*(C_j), and it works as follows in each round:
31: t joins into H by $\mathcal{I}$-join except line 1,2, and each neighbor v of t adds t to *v.link*;
32: t sends the token back to *lev* by *Eulerian*(T');
33: nodes who have joined to H are removed from T';
34: **end while**
35: **while** there is a *lev*'s child $v \in C_i$ who has not determined its status **do** {separation}
36: *lev* sends a message to v;
37: v makes $CNet(C_i)$ with root v by *Eulerian*(C_i) performing $\mathcal{I}$-join one by one;
38: **end while**
39: **end if**

Procedure 7 subroutine of $\mathcal{I}$-leave

exception

1: *lev* sends a message to one of its neighbors r, and r becomes the root and has a token ($t := r$);
2: r sends *I'mRoot* message and $v.link := v.link \cup \{r\}$ for each neighbor v of r;
3: % Let $G'' := G[\{r\}]$;
4: t calls $Eulerian(G[G(V) - \{lev\}])$ which works as follows in each round:
5: **if** t has not joined **then**
6: t joins into G'' according to the status of nodes in $t.link$ by $\mathcal{I}$-join;
7: Each neighbor v' of t adds t to $v'.link$;
8: **end if**

change-root

1: *lev* sends a message to a head h in $lev.bneigh$;
2: $lev.prt := h$, $h.chd := h.chd \cup \{lev\}$;
3: h sends a message to a head h' in $h.bneigh$;
4: $h.prt := h'$, $h'.chd := h'.chd \cup \{h\}$;
5: $h'.prt := \perp$ and h' becomes a root;

Procedure 8 $\mathcal{U}$-status

1: **if** there are heads in neighbors of *new* **then**
2: % Let h be a head with minimum ID in neighbors of *new*;
3: $new.stat :=$ member2, $new.prt := h$, $h.chd := h.chd \cup \{new\}$;
4: **else if** there are member1s in neighbors of *new* **then**
5: % Let m_1 be a member1 with minimum ID in neighbors of *new*;
6: $new.stat :=$ member2, $m_1.stat :=$ head;
7: $new.prt := m_1$, $m_1.chd := m_1.chd \cup \{new\}$;
8: **else** {only member2}
9: % Let m_2 be a member2 with minimum ID in neighbors of *new*;
10: $new.stat :=$ member1, $m_2.stat :=$ head;
11: $new.prt := m_2$, $m_2.chd := m_2.chd \cup \{new\}$;
12: **end if**

Procedure 9 $\mathcal{U}$-merge

1: % Let r be the minimum rootID in neighbors of *new*;
2: % Let $R = G[\{v | v \in CNet_{\mathcal{I}}$ with rootID $r\} \cup \{new\} \cup \{v | v$ has called $\mathcal{I}$-status$\}]$;
3: % Let t be a node with a token in *Eulerian*;
4: *new* call $\mathcal{U}$-status for $CNet_{\mathcal{U}}$ with rootID r;
5: **for** each rootID i of *new*'s neighbors **do**
6: **if** there are heads with rootID i in neighbors of *new* **then**
7: *new* sends *DoJoin* message to a neighboring head u with rootID i and minimum ID;
8: **else if** there are member1s with rootID i in neighbors of *new* **then**
9: *new* sends *DoJoin* message to a neighboring member1 u with rootID i and minimum ID;
10: **else**
11: *new* sends *DoJoin* message to a neighboring member2 u with rootID i and minimum ID;
12: **end if**
13: u call $\mathcal{U}$-status for R;
14: u calls *Eulerian* on the backbone with rootID i which works as follows in each round:
15: t joins into R by $\mathcal{U}$-status;
16: **if** t becomes member **then**
17: t send message to its children, and they set their status into member2;
18: *deliver-member*(t);
19: **end if**
20: Nodes in $t.chd$ set their parent into t;
21: **end for**

by $\mathcal{I}$-join/leave.

Lemma 11. *Let $G = (V, E)$ be a unit disk graph, and $MDS(G)$ be the minimum dominating set of G. $|BT_{\mathcal{I}}(G)|$ is not larger than $10 \times |MDS(G)|$.*

The properties shown in lemmas 10, 11 are not derived from the definition of structure of $\mathcal{I}$, but derived from the maintenance algorithm (Algorithm 2, 6).

On $\mathcal{I}$, a merge can be performed without changing the order of completion time for broadcast as well as $\mathcal{M}$. And in the case that a merge is not caused, the size of a backbone tree and the completion times of join/leave algorithms are the same as [2]. Kinds of status is reduced and a backbone tree is simplified on $\mathcal{I}$.

3.6 $\mathcal{U}$

We show the last architecture $\mathcal{U}$. On $\mathcal{U}$, the same theoretical results on $\mathcal{I}$ are obtained, e.g. the completion time of operations and the size of backbone tree.

3.6.1 Structure

$\mathcal{U}$ follows the basic structure of $\mathcal{I}$, and attempt to reduce the number of clusters (i.e., the size of backbone tree) by dividing member into two status.

In the structure, a set of heads is not independent set of G and the backbone tree consists of only heads as well as $\mathcal{I}$. We use member1 and member2 as status except for heads.

Let $CNet_{\mathcal{U}}(G)$ and $BT_{\mathcal{U}}(G)$ be $CNet(G)$ and $BT(G)$ with architecture $\mathcal{U}$, respectively.

3.6.2 Maintenance Algorithms

First we show the join algorithm for $\mathcal{U}$.

We extract only a part relative to a determination of status and merge operation in Procedure 8 and 9. By replacing the name $\mathcal{I}$ with the name $\mathcal{U}$, most part of the other algorithm and procedure of $\mathcal{U}$ are the same as that of $\mathcal{I}$.

Theorem 4. *For disjoint graphs $G_1, G_2, \ldots$ and $CNet_{\mathcal{U}}(G_1), CNet_{\mathcal{U}}(G_2), \ldots$, when G is organized with total 1-hop data, $\mathcal{U}$-join can be done in $O(q)$ expected rounds with no merge or in $O(q + \max\{|BT(G)|$ for each $CNet\})$ expected rounds with merge, and G' is organized with total 1-hop data, where q is the number of neighbors of new in G'.*

In the execution of $\mathcal{U}$-join only the neighbors of *new*, and neighbors of the neighbors of *new* in $CNet_{\mathcal{U}}(G)$ receive a message caused by join. Hence, join can be performed locally in only 2-hop neighbors of *new* without changing status of other nodes.

About our leave algorithm to maintain $\mathcal{U}$, it is basically the same as $\mathcal{I}$-leave, where we consider member1 and member2 as member and $\mathcal{I}$-join as $\mathcal{U}$-join in $\mathcal{I}$-leave

algorithm. Only in the case that the leaving node *lev* is a root or member1, the algorithm differs slightly.

If *lev* is member1, *lev* changes its status into head, and performs $\mathcal{I}$-leave. Else if *lev* is a root, it searches a head 2-hop away. If such a head does not exist, it searches a member1 2-hop away, changes its status into head, and performs $\mathcal{I}$-leave. Else it simply performs $\mathcal{I}$-leave.

Theorem 5. *Let $CNet_{\mathcal{U}}(G)$ be a cluster-based network of G and T be the subtree of $CNet_{\mathcal{U}}(G)$ with the leaving node lev as root. When G is organized with total 1-hop data, $\mathcal{U}$-leave can be done in $O(|T|)$ rounds, and G' is organized with total 1-hop data.*

3.6.3 Properties

$\mathcal{U}$ which is constructed above two operations has the following property.

Lemma 12. *The number of clusters is equal to $|BT_{\mathcal{U}}(G)|$.*

Lemma 13. *Let G be a connected bidirectional graph and p_G be the cardinality of minimum clique partition of G. $|BT_{\mathcal{U}}(G)|$ is at most $2p_G$ on $\mathcal{U}$ which is constructed by $\mathcal{U}$-join/leave.*

Lemma 14. *Let $G = (V, E)$ be a unit disk graph, and $MDS(G)$ be the minimum dominating set of G. $|BT_{\mathcal{U}}(G)|$ is not larger than $10 \times |MDS(G)|$.*

As well as $\mathcal{I}$, the properties shown in Lemmas 13, 14 are not derived from the definition of structure of $\mathcal{U}$, but derived from the maintenance algorithm.

$\mathcal{I}$ satisfy the same properties of $\mathcal{M}$ and $\mathcal{I}$ about the capability of merge / the completion time of broadcast / the size of back bone tree. We more show the advantage of the size of backbone tree on $\mathcal{U}$ by a simulation in the next section.

4. Simulation

We compare the sizes of backbones for the four architectures and its maintenance algorithms by simulation. The setting of the simulation is as follows: Each node is treated as a point without volume; The field where nodes are deployed is infinite plane; Nodes are added to the field from initial state with one node until the number of nodes reach n; Each node is setted randomly within a range where existing nodes can tranmit.

Under above setting, we measure the size of backbone within the limit of $n = 1000, \ldots, 8000$ by 1000 nodes (Fig. 3). Each plot point represent a average value for 100 trials.

The size of backbone tree of $\mathcal{U}$ is less than that of the other architectures within the limit of $n = 1000, \ldots, 8000$, and it is expected that $\mathcal{U}$ is also superior to the others for $n \gg 8000$ from Fig. 3.

We can consider that the size of backbone tree of $\mathcal{I}$ is slightly less than that of $\mathcal{C}$ and $\mathcal{M}$ for the reason that the number of nodes which can exist as member increase by constructing backbone tree with only heads, i.e. the nodes that are able to have children. It is also considered that $\mathcal{U}$ can greatly decrease the number of heads which exist in a part of the outer of the region where nodes are deployed actually owing to m_1.

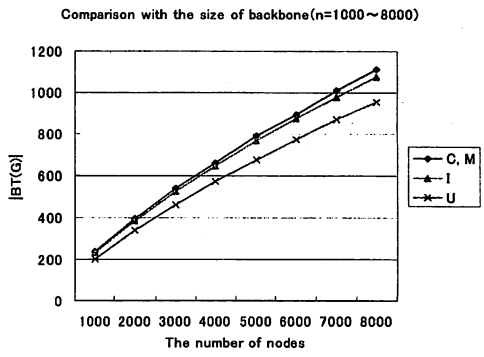

Fig. 3 Comparison with the size of backbone tree

5. Conclusions

In this paper we improve the architecture [2], and propose the architectures and the maintenance algorithms which have better properties about the size of a backbone, expansions of tasks, and so on. Also we enable the merge/separation operation which is taken no account of in [2].

References

[1] S. Basagni, M. Mastrogiovanni, C. Petrioli, A performance comparison of protocols for clustering and backbone formation in large scale ad hoc networks, *The 1st Internatinal Conderence on Mobile Ad-hoc and Sensor Systems*, pp.70-79, 2005.

[2] J. Uchida, M. A.K.M. Islam, Y. Katayama, W. Chen, and K. Wada, Construction and Maintenance of a Cluster-based Architecture for Sensor Networks. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06) Track 9*, p. 237c, 2006.

[3] S. Miyanaga, Y. Katayama, K. Wada, N. Takahashi, M. Kobayashi, M. Morita On Efficient Clustering Algorithms for Dynamic Wireless Ad-hoc Networks with Considering Mergence and Partition of Clusters. *IEICE Technical Report*, vol.105, no.679, COMP2005-58, pp. 9–16.

[4] B.S. Chlebus, L. Gąsieniec, A.M. Gibbons, A. Pelc, and W. Rytter. Deterministic broadcasting in ad hoc radio networks. *Distributed Computing 15*, pages 27–38, 2002.

[5] R. Bar-Yehuda, O. Goldreich, and A. Itai, On the time-complexity of broadcast in radio networks: an exponential gap between determinism and randomization, *Journal of Computer and System Science*, no. 45, pp. 104–126, 1992.

[6] K. Nakano and S. Olariu. Randomized initialization protocols for radio networks. *Handbook of wireless networks and mobile computing*, pp.195–218, 2002.

[7] J. Uchida, M. A.K.M. Islam, Y. Katayama, W. Chen, and K. Wada, Toward better cluster-based architectures for Ad hoc sensor networks. *Technical report*, Dept. of Computer Science and Engineering, Nagoya Institute of Technology, 2006.