Distributed Hierarchy of Clusters in the Presence of Topological Changes

François Avril, Alain Bui, and Devan Sohier

Laboratoire PRiSM (UMR CNRS 8144), Université de Versailles, France
francois.avril@uvsq.fr, {alain.bui,devan.sohier}@prism.uvsq.fr

Abstract. We propose an algorithm that builds a hierarchical clustering in a network, in the presence of topological changes. Clusters are built and maintained by random walks, that collect and dispatch information to ensure the consistency of clusters.

We implement distributed communication primitives allowing clusters to emulate nodes of an overlay distributed system. Each cluster behaves like a virtual node, and executes the upper level algorithm. Those primitives ensure that messages sent by a cluster are received and treated atomically only once by their recipient, even in the presence of topological changes. Decisions concerning the behavior of the cluster (virtual node for the higher level algorithm) are taken by the node that owns the random walk at this time.

Based on this abstraction layer and the overlay network it defines, we present a distributed hierarchical clustering algorithm, aimed at clustering large-scale dynamic networks.

1 Introduction

We deal in this paper with the problem of clustering large dynamic networks. This study is motivated by the need to cope with topological changes in a local and efficient manner: as far as possible, we intend to take in charge a topological change inside the cluster where it happened, transparently for the rest of the network.

To achieve this, the clustering algorithm itself must be resilient to topological changes. Thus, after a topological change, the clustering is recomputed only locally: the only clusters affected by the algorithm are the cluster in which the topological change occurred and possibly adjacent clusters.

Based on the algorithm presented in [4], that computes clusters with a size bounded by a parameter K of the algorithm, we propose a distributed algorithm above this clustering, by making each cluster behave as a virtual node. We then apply this method to build a distributed hierarchical clustering.

The cluster is embodied in a special message, the *Token*: any action taken by the cluster is actually initiated by the node that holds the token, making the treatment of higher level messages atomic.

H.A. Le Thi et al. (eds.), *Advanced Computational Methods for Knowledge Engineering*,
Advances in Intelligent Systems and Computing 358, DOI: 10.1007/978-3-319-17996-4_33

The inter-cluster communication primitive *SSend* is implemented so as to ensure that it works even in the presence of topological changes: if a message is sent by a cluster to an adjacent cluster using this primitive, it is eventually received if the clusters remain adjacent. Then the associated treatment is executed exactly once. Thus, the mobility of nodes affects only a limited portion of the system.

A distributed algorithm can then be applied on the overlay network defined by the clustering. If this algorithm is resistant to topological changes, the resulting method will also be resistant to topological changes. In particular, we apply this method to the presented clustering, leading to a distributed bottom-up hierarchical clustering algorithm.

1.1 Related Works

Designing distributed algorithms that build a nested hierarchical clustering has recently been the subject of many research works (see for example [7]). Those methods build a clustering with a standard method iterated on clusters, represented by a distinguished node called the clusterhead. First, clusterheads are elected in the network. Then, each clusterhead recruits its neighbors. This election can be done on several criteria, such as their id ([1]), their degree ([8]), mobility of nodes ([3]), a combination of those criteria ([5]) or any kind of weight ([2],[9]). The algorithm is then iterated on the graph of the clusters, involving only clusterheads. This results in creating clusters of clusters, and thus building a hierarchical nested clustering by a bottom-up approach.

These methods are very sensitive to topological changes: after a topological change, all clusters can be destroyed before a new clustering is computed, with possibly new clusterheads.

In this paper, we present an algorithm based on the clustering in [4]: a fully decentralized one-level clustering method based on random walks, without clusterhead election, and with the guarantee that reclusterization after a topological change is local, i.e. after a cluster deletion, only clusters adjacent to this cluster can be affected. We design a hierarchical clustering by allowing every cluster in every level to emulate a virtual node. This algorithm preserves properties of [4].

Previous works on virtual nodes, like [6] or [10], aim at simplifying the design of distributed algorithms. Virtual nodes are in a predetermined area of the network, and their mobility is known in advance. Several replica of the virtual machine are necessary to resist topological changes and crashes. Maintaining coherence between replica requires synchronized clocks and a total order on messages, which also allows to verify FIFO hypothesis on channels.

In this work, virtual nodes are built dynamically, with weaker hypotheses on communications: non-FIFO, asynchronous.

1.2 Model and Problem Statement

We suppose that all nodes have a unique id, and know the ids of their neighbors. In the following, we do not distinguish between a node and its id.

The distributed system is modeled by an undirected graph $G = (V, E)$. V is the set of nodes with $|V| = n$. E is the set of bidirectional communication links. The neighborhood of a node i (denoted by N_i) is the set of all nodes that can send messages to and receive messages from i.

We make no assumption on the delivery order of messages. Indeed, the protocol we define for communications between virtual nodes does not verify any FIFO property.

We suppose that all messages are received within finite yet unbounded time, except possibly when the link along which the message is sent has disappeared because of a topological change.

The algorithm is homogeneous and decentralized. In particular, all nodes use the same variables. If *var* is such a variable, we note var_i the instance of *var* owned by node i.

A random walk is a sequence of nodes of G visited by a token that starts at a node and visits other vertices according to the following transition rule: if a token is at i at time t then, at time $t + 1$, it will be at one of the neighbors of i chosen uniformly at random among all of them.

All nodes have a variable *cluster* that designates the cluster to which they belong. A boolean *core* indicates if the node is in the cluster core. All nodes know a parameter K of the algorithm, that represents an upper bound on the size of a cluster.

Definition 1. *The cluster C_x is the set of nodes $\{i \in V / cluster_i = x\}$. Its core is the set $K_x = \{i \in C_x / core_i\}$. A cluster is said complete if $|K_x| = K$.*

The clustering we are computing is aimed at giving clusters with cores of size K as far as possible. To ensure intra-cluster communications, a spanning tree of each cluster is maintained, through a *tree* variable, with $tree[i]$ the father of i in the tree.

Definition 2 (Specification). *A cluster is called consistent if: it is connected; its core is a connected dominating set of the cluster; its core has a size between 2 and K with K a parameter of the algorithm; a single token circulates in the cluster and carries a spanning tree of the cluster.*

A clustering is called correct if: all clusters are consistent; each node is in a cluster; a cluster neighboring an incomplete cluster is complete.

This, together with the property that an incomplete cluster has no ordinary node, guarantees that clusters have maximal cores with respect to their neighborhood.

The algorithm presented next eventually leads to a correct clustering, after a convergence phase during which no topological change occurs. Each topological change may entail a new convergence phase.

Topological changes result in configurations in which the values of variables are not consistent with the actual topology of the system. Then, mechanisms triggered by nodes adjacent to the topological change allow to recompute correct clusters, without affecting clusters other than the one in which the change occurred, and possibly adjacent clusters.

Additionally, we require that primitives are defined for inter-cluster communications, providing the functionalities of *send* and *upon reception of* to higher levels. The $SSend$ primitive we define and implement ensures that a node holding the token can send higher level messages to adjacent clusters. Then, we guarantee that the token in charge of the recipient cluster eventually meets this message, and executes the appropriate treatment on the variables it stores. This allows to implement a distributed algorithm on the clustering, with clusters acting as nodes w.r.t. this algorithm.

We define a hierarchical clustering, in which a correct clustering of level i is a correct clustering on the overlay graph defined by the clusters of level $i-1$, with edges joining adjacent clusters.

2 Algorithm

2.1 Clustering

Each unclustered node may create a cluster, by sending a token message to a neighbor chosen at random. The token message then follows a random walk, and recruits unclustered and ordinary nodes it visits to the cluster core. It visits all nodes of the cluster infinitely often, which ensures the updating of variables on nodes and in the token message. When a token message visits a node in another cluster core, it may initiate the destruction of its own cluster with a *Delete* message if both clusters are incomplete and if the id of the token cluster is less than that of the visited cluster. Thus two adjacent clusters cannot be both incomplete once the algorithm has converged.

A node that receives a *Delete* message from its father leaves its cluster and informs all its neighbors. This initiates the destruction of the cluster by propagation of *Delete* messages along the spanning tree.

All nodes periodically inform their neighbors of their cluster by sending a $Cluster[cluster]$ message, which ensures the coherence of *gate* variables.

Each node knows the cluster to which it belongs with the variable *cluster*; it knows if it is a core node thanks to the *core* variable. Some other technical variables are present on each node : *complete* indicates if the cluster to which the node belongs is complete, $father$ is the father of the node in the spanning tree.

The algorithm uses five types of messages: $Token[id, topology, N, gateway, Status]$, $Recruit[id]$, $Cluster[id]$, $Delete[id]$ and $Transmit[message, path, em, dst]$.

Token messages carry the following variables: *id*, the id of their cluster *id*; N, a list of adjacent clusters; *tree*, a spanning tree of the cluster; *size*, the size of the cluster core; *corenodes*, an array of booleans indicating which nodes are in the cluster core, *gateway*, a vector of links to adjacent clusters; and *status*, a structure containing the same variables as a node, to be used by upper levels.

Token messages follow a random walk and recruit core nodes. $Recruit[id]$ messages recruit nodes adjacent to a core node as ordinary nodes in cluster *id*,

Delete[*id*] messages delete the cluster *id* and *Cluster*[*id*] messages inform that the sender is in cluster *id*.

A variable *var* on a message *Token* is noted *token.var*.

Initialization: When a node connects to the network, we assume that all variables are set to 0 or $\perp$, except for T, that is initialized with a random value.

JoinCore **function:** The *JoinCore* function is used to update variables in node i when the node enters a cluster as a core node.

Algorithm 1. *JoinCore*()

$cluster \leftarrow token.id$
$core \leftarrow true$
$token.size \leftarrow token.size + 1$
$token.corenodes[id] \leftarrow true$
$complete \leftarrow (token.size \geq K)$

On Timeout. When the timeout on node i is over, it creates a new cluster, with id $x = (i, nexto)$, and increments *nexto*. Node i joins cluster x and then sends a token message to a neighbor chosen uniformly at random. This node becomes the father of i in the spanning tree. Node i also recruits all its unclustered neighbors as ordinary nodes by sending them *Recruit* messages.

Algorithm 2. *timeout*()

if $N \neq \emptyset$ **then**
  $token$ =new token message
  $token.size \leftarrow 0$
  $token.id \leftarrow (id, nexto)$
  $token.tree \leftarrow$ EmptyVector
  $token.tree[id] \leftarrow id$
  UpdateStatus()
  JoinCore()
  $nexto \leftarrow nexto + 1$
  Send *Cluster*[*cluster*] to all nodes in N
  Send *Recruit*[*cluster*] to all nodes in N
  $father \leftarrow$ random value in N
  Send *token* to $father$
else
  $T \leftarrow$ random value

On Reception of a *Recruit*[*id*] **Message.** On reception of a *Recruit*[*id*] message, if node i is unclustered, it joins cluster *id* as an ordinary node by calling *JoinOrdinary*. The core node that sent the *Recruit* message becomes the father of i in the spanning tree.

Algorithm 3. *JoinOrdinary*(*Sender*)

if $(cluster = (\perp, 0)) \vee (cluster = recruit.id)$ **then**
  $cluster \leftarrow recruit.id$
  $core \leftarrow false$
  $father \leftarrow Sender$
  Send *Cluster*[*cluster*] to all nodes in N

On Reception of a *Delete*[*id*] **Message from** e**.** When node i receives a *Delete*[*id*] message from e, if e is its father in the spanning tree, i leaves the cluster and sends a *Delete*[*id*] message to all its neighbors.

Algorithm 4. On reception of a $Delete[id]$ message from e

if $e = father \wedge delete.id = cluster$ **then** Send $Delete[cluster]$ to all neighbors.
  $LeaveCluster()$

$LeaveCluster()$ reinitializes all variables and sends a $Cluster[(\perp, 0)]$ message to all neighbors. *Delete* messages are then propagated along the cluster spanning tree.

On Reception of a $Cluster[id]$ Message from e. When node i receives a $Cluster[id]$ message from node e, it stores the received cluster id in $gate[e]$.

On j Leaving N (Disconnection between Current Node and a Neighbor j). The disappearance of an edge (i, j) can lead to an unconnected (and thus faulty) cluster only if the link is in the spanning tree. If a node loses connection with its father in the spanning tree, it leaves the cluster by calling $LeaveCluser()$.

When a communication link (i, j) disappears, messages in transit on this link may be lost. A $Recruit[id]$ or a $Cluster[id]$ message is no longer necessary, since nodes are no longer adjacent. If a $Delete[id]$ message is lost, the system will react as if it receives it when j detects the loss of connection. If a $Token$ message is lost, the associated cluster is deleted: indeed, if i had sent a $Token$ message to j, it considers j as its father. When i detects the loss of connection, it initiates the destruction of the cluster. Last, if a $Transmit$ message is lost, then this is a topological change at the upper level, and the same arguments apply.

In any case, the configuration resulting from the loss of a message on a disappeared channel may be the result of an execution without any message loss, and the algorithm continues its operation transparently.

On Reception of a $Token[id, topology, N, gateway, status]$ Message. When node i receives a $Token[id, topology, N, gateway, status]$ message from node e, the following cases may occur:

Update Information on a Core Node: If i is a core node of cluster *token.id*, variables *token.topology* and *complete* are updated. Then, the treatment of upper level messages is triggered. Then, i sends $Recruit[$ *token.id*$]$ messages and $Cluster[$*token.id*$]$ messages to all its neighbors, and sends the token to a neighbor chosen at random.

Algorithm 5. $token(Sender)$ on a node i with $(cluster_i = token.id) \wedge (core_i)$

{update information - core node}
$UpdateStatus()$
if $(token.tree[i] \neq i)$ **then**
  {the token has not just been bounced back by another cluster}
  $token.tree[e] \leftarrow id$
  $token.tree[id] \leftarrow id$
$complete \leftarrow (token.size \geq K)$
TriggerUpperLevel()
Send $Recruit[token.id]$ to all nodes in N
Send $Cluster[token.id]$ to all nodes in N
Choose $father$ uniformly at random in N
Send $Token[id, topo, N, gateway, status]$ to $father$

Update Information on an Ordinary Node: If i is an ordinary node of cluster $token.id$ and the cluster is complete ($token.size = K$), i cannot be recruited. Information on token is updated, upper level messages are processed and the token is sent back to the core node that sent it.

Algorithm 6. $token(Sender)$ on a node i with $(cluster_i = token.id) \wedge (\neg core_i) \wedge (token.size \geq K)$

{update information - ordinary node}	TriggerUpperLevel()
$UpdateStatus()$	$father \leftarrow e$
$token.tree[e] \leftarrow id$	Send $Token[id, topo, N, gateway, status]$
$token.tree[id] \leftarrow id$	to $father$

Recruit a Node to the Core: If i is an ordinary or unclustered node, it becomes a core node of cluster $token.id$. Information on the token is updated, upper level messages are processed, $Recruit[cluster]$ and $Cluster[cluster]$ messages are sent to all neighbors, and the token is sent to a neighbor chosen uniformly at random.

Algorithm 7. $token(Sender)$ on a node i with $(token.size < K) \wedge (\neg core_i)$

{recruit a node to the core}	$complete \leftarrow (token.size \geq K)$
if $cluster \neq token.id$ **then**	$TriggerUpperLevel()$
LeaveCluster()	Send $Cluster[cluster]$ to all nodes in N
$JoinCore()$	Send $Recruit[cluster]$ to all nodes in N
$UpdateStatus()$	Choose $father$ uniformly at random in N
$token.tree[e] \leftarrow id$	Send $Token[id, topo, N, gateway, status]$
$token.tree[id] \leftarrow id$	to $father$

Initiate the Destruction of the Cluster: A cluster x can destroy a cluster y when y has a size of 1, or when both clusters are non-complete and the id of x is greater than that of y. If the token cluster can be destroyed by the node cluster, the token message is destroyed and a $Delete[cluster]$ and a $Recruit[cluster]$ messages are sent to the sender, that considers i as its father in the spanning tree. This leads to the destruction of cluster $token.id$.

Algorithm 8. $token(Sender)$ on a node i with $[(token.size < K) \wedge (\neg complete_i) \wedge (cluster_i > token.id)] \vee (token.size = 1)$

{initiate the destruction of the cluster $token.id$}	Send $Delete[token.id]$ to e
	Send $Recruit[cluster]$ to e

Send the Token Back: In all other cases, the token message is sent back to the sender.

Algorithm 9. *token*(*Sender*) on all other cases

{send the token back}
if $token.tree[id] \neq id$ **then**
  Send $Token[id, topo, N, gateway, status]$ to e

Algorithm 10. *UpdateStatus*()

for all k with $token.tree[k] = i$ {i the id of the node runing this function} **do**
  if $k \notin N$ **then**
    $RemoveFromTree[k, token.topology]$
for all id with $token.gateway[id] \neq \perp$ **do**
  $(l, m) \leftarrow token.gateway[id]$
  if $(l \notin token.tree) \vee [(l = i) \wedge ((m \notin N) \vee (gate_i[m] \neq id))]$ **then**
    $token.gateway[id] \leftarrow \perp$
    remove id from $token.N$
    {This may entail a topological change w.r.t. the higher level, in which case, it triggers the higher level *Leave* function}
for all $k \in N$ with $gate[k] \neq (\perp, 0)$ and $cluster > gate[k]$ **do**
  $token.gateway[gate[k]] \leftarrow (i, k)$
  Add $gate[k]$ to $token.N$

2.2 Virtual Nodes

We aim at making every cluster behave as a virtual node, in order to be able to execute a distributed algorithm on the overlay network defined by the clustering. In particular, this will allow to build a hierarchical clustering through a bottom-up approach. In this section, we present mechanisms that allow a cluster to mimic the behavior of a node:

- virtual nodes know their neighbors (adjacent clusters), and are able to communicate with them;
- virtual nodes execute atomically the upper level algorithm.

To ensure that clusters have a unique atomic behavior, the node that holds the *token* message is in charge of the decision process about the cluster seen as a virtual node. Variables of the virtual node are stored in $token_{id}.Status$.

Knowledge on the neighborhood is maintained by *Cluster* messages and *UpdatesStatus* function, as seen in section 2.1.

Neighborhood Observation. Since $Cluster[id]$ messages are sent infinitely often, all nodes maintain the vector *gate* indicating the cluster to which each of their neighbors belong. When a message $token_x$ visits a node in cluster x that has a neighbor in another cluster y, the link between the two nodes can be selected as a privileged inter-cluster communication link between clusters x and y, and added to $token_x.gateway$ and $token_y.gateway$. To ensure that topological changes are detected symmetrically at upper level, adjacent clusters need to agree on the gateway they use to communicate, i.e. $token_{C_x}.gateway[C_y] = token_{C_y}.gateway[C_x]$. Thus, when this link breaks, both clusters are aware of this loss of connexion between the two virtual nodes, which allows the overlay graph to remain undirected. Only the cluster with the smallest id can select an inter-cluster communication link between two adjacent clusters. The other cluster adds it at the reception of the first upper level message.

The *gateway* array in $Token$ message, along with the spanning tree stored in the $Token$ message, allows to compute a path from the node holding the $Token$ to a node in any adjacent cluster. This enables virtual nodes to send upper level messages to their neighbors: see Algorithm 11.

Thus, if $token.gateway[id] = (i, j)$, then j is a node in cluster id. To send a message to an adjacent cluster id, $SSend$ computes a path to j in the spanning tree stored in $token.topology$ and adds link (i, j). Then, the upper level message is encapsulated in a $Transmit$ message, and routed along this path.

Communication. $SSend$ is used when handling a topological change, or when upper level messages are received. In both cases, this treatment can be initiated only in presence of the token. Thus, $token.topology$ information is available. On a higher level, the processing of a $Token$ message is triggered by the processing of the lower level token. So, when processing a token message, all lower level tokens are available.

Algorithm 11. $SSend(msg, dst, tok)$ function

$(l, m) \leftarrow tok.gateway[dst]$	$lowlevmsg \leftarrow Transmit[msg, Tail(path),$
$path \leftarrow ComputePath(tok.topo, l)$	$tok.id, dst]$
$path \leftarrow (path, m)$	$SSend(lowlevmsg, Head(path), token)$

The recursive calls to $SSend$ end with the bottom level using $Send$ instead of $SSend$.

Upper level messages are encapsulated in $Transmit$ messages, along with the path to follow, sender and recipient virtual nodes id. Such an upper level message is forwarded along the path to a node of the recipient cluster. This node stores three pieces of information in $listmsg$: the encapsulated message; the node that sent the $Transmit$ message; the cluster that sent the encapsulated message.

This information ensures that the cluster can reply to the sender cluster; it will be used to add a link to this adjacent cluster and guarantee the symmetry of comunications. When the token visits a node in this cluster, it triggers the processing of upper level messages stored in $listmsg$ by the virtual node, and removes them from $listmsg$.

Algorithm 12. Reception of a $Transmit[msg, path, em, dst]$ message from e

if $(path = \emptyset) \wedge (cluster = dst)$ **then**	Send $Transmit[msg, Tail(path), em,$
Add (msg, e, em) to $listmsg$	$dst]$ to $Head(path)$
else if $((cluster = em) \wedge (Head(path) \in$	
$N))$ **then**	

Processing the Upper Level. Processing of upper level messages stored in a node is triggered when the $Token$ message visits the node. Function

TriggerUpperLevel then pops messages out of *listmsg* and emulates the reception of this message by the virtual node. *TriggerUpperLevel* also manages the triggering of function *timeout* for upper level, simulating a timer by counting *Token* hops.

If the sender virtual node is not in *token.N* and *token.gateway*, *listmsg* contains enough information to add it. Once all upper level messages are treated, it decrements the count-down on *token.T* in structure *token.status*.

Algorithm 13. *TriggerUpperLevel*()

```
while (listmsg ≠ ∅) do
  (msg, s, C) ← Pop(listmsg)
  if token.gateway[C] = ⊥ then
    token.gateway[C] ← (i, s)
    Add C to token.N
  else if token.gateway[C] ≠ (i, s)
  then
    run "loss of connection" procedure
    on the virtual node
  token.gateway[C] ← (i, s)
  Add C to token.N
Emulate reception of msg by the vir-
tual node using SSend.
if (token.T > 0) then
  token.T ← token.T − 1
  if (token.T = 0) then
    Stimeout()
```

Functions for processing upper levels are identical to the first level functions presented in subsection 2.1, except that they use *SSend* instead of *send*.

2.3 Hierarchical Clustering

If the upper level algorithm executed by the virtual nodes is this same clustering algorithm, the virtual nodes compute a distributed clustering resistant to topological changes. Iterating this process, we obtain a hierarchical clustering.

This provides a framework to contain the effect of topological changes inside clusters, while allowing communication between nodes in any two clusters. In particular, a modular spanning tree of the system consisting in the spanning trees of clusters of all levels is built.

3 Example

We present here an example involving three levels of clustering. Consider a clustering with C_A (a cluster of level ≥ 3) containing clusters C_x and C_y, C_B a cluster of the same level as C_A, containing C_z. C_x, C_y and C_z are composed of nodes or of clusters of a lower level. Consider that cluster C_A sends a $Cluster[(\perp, 0)]$ message to cluster C_B.

Sending the Message Out of x. C_A sends a $Cluster[(\perp, 0)]$ message to C_B. This is triggered by the processing of the $token_{C_A}$ message on a lower level. Thus, cluster C_x necessarily holds the $token_{C_A}$ message. It calls the *SSend* mechanism and uses information on $token_{C_A}$ to compute a path to C_B, here to cluster z (see figure 1). Then, since the path is xyz, cluster C_x sends a $Transmit[Cluster[(\perp, 0)], z, A, B]$ to the next cluster on the path, C_y (with

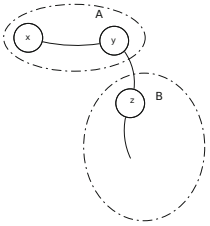

Fig. 1. Seen from $token_{C_A}$

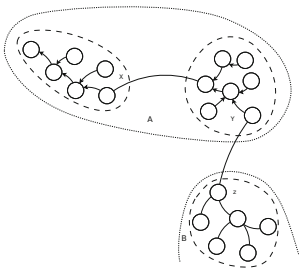

Fig. 2. Detailed view

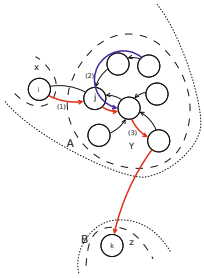

Fig. 3. Routing in C_y

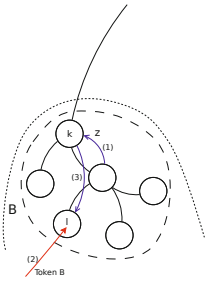

Fig. 4. Reception of the message

the *SSend* mechanism). A node of C_y finally receives a *Transmit* message: $Transmit[Transmit[Cluster[(\perp, 0)], z, A, B], \emptyset, x, y]$.

Routing the Message in C_y.

1. Node j in C_y receives a $Transmit[Transmit[Cluster[\ (\perp, 0)], z, A, B], \emptyset, x, y]$ from node i. Since it is in C_y, and the path is empty, it stores the transmitted message $(Transmit[Cluster[\ (\perp, 0)], z, A, B], i, y)$ in its *listmsg*.
2. $token_{C_y}$ eventually visits j and treats upper level messages. Since $path = z \neq \emptyset$, node j uses *SSend* to send $Transmit[Cluster[(\perp, 0)], \emptyset, A, B]$ to cluster C_z (line 5 algorithm 12). *SSend* uses $token_{C_y}.topology$ to compute a path to $token_{C_y}.gateway[z]$ and sends a *Transmit* message to $head(path)$: $Transmit[\ Transmit[\ Cluster[(\perp, 0)], z, A, B], tail(path), y, z]$.
3. Every node in the path transmits the *Transmit* message until node k receives the message $Transmit[\ Transmit[\ Cluster[(\perp, 0)], \emptyset, A, B], \emptyset, y, z]$.

Reception of the Message by C_B. Node k receives a *Transmit* message from node e $(Transmit[\ Transmit[\ Cluster[(\perp, 0)], \emptyset, A, B], \emptyset, y, z])$ and stores $(Transmit[Cluster[(\perp, 0)], \emptyset, A, B], e, y)$ in $listmsg$ (algorithm 12, line 2). Then:

1. $token_{C_z}$ eventually visits k during its random walk, and treats messages in $listmsg_k$. $(Cluster[(\perp, 0)], y, A)$ is stored in $token_{C_z}.listmsg$.
2. $token_{C_B}$ eventually visits cluster C_z during its random walk, i.e. it is eventually stored in the $listmsg_l$ variable of a node l in C_z.
3. $token_{C_z}$ eventually visits node l. Then, it processes $token_{C_B}$. *TriggerUpperLevel* is called and messages in $token_{C_Z}.listmsg$ are treated. In particular, $(Cluster[\ (\perp, 0)], y, A)$ is processed, ie $token_{C_B}.gate[A] \leftarrow (\perp, 0)$.

4 Conclusion and Perspectives

The algorithm presented in this paper computes a size-oriented hierarchical clustering of a dynamic network. It reacts to topological changes in a local manner:

after a topological change that makes the clustering faulty, the only clusters affected are the cluster in which this event took place and possibly some adjacent clusters.

Thus, it is designed to provide a local mechanism to handle topological changes. Inter-cluster communication mechanisms are proposed that allow to implement a distributed algorithm above the clustering, and a formal proof is provided.

Proofs of this algorithm may be found at: `http://www.prism.uvsq.fr/~sode/hierclus/proof.pdf`.

References

1. Baker, D., Ephremides, A.: The architectural organization of a mobile radio network via a distributed algorithm. IEEE Transactions on Communications 29(11), 1694–1701 (1981)
2. Basagni, S.: Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In: IEEE VTS 50th Vehicular Technology Conference, VTC 1999 - Fall, vol. 2, pp. 889–893 (1999)
3. Basagni, S.: Distributed clustering for ad hoc networks. In: Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 1999), pp. 310–315 (1999)
4. Bui, A., Kudireti, A., Sohier, D.: An adaptive random walk-based distributed clustering algorithm. International Journal of Foundations on Computer Science 23(4), 802–830 (2012)
5. Chatterjee, M., Das, S.K., Turgut, D.: Wca: A weighted clustering algorithm for mobile ad hoc networks. Journal of Cluster Computing (Special Issue on Mobile Ad hoc Networks) 5, 193–204 (2001)
6. Dolev, S., Gilbert, S., Lynch, N.A., Schiller, E., Shvartsman, A.A., Welch, J.L.: Virtual mobile nodes for mobile *ad hoc* networks. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 230–244. Springer, Heidelberg (2004)
7. Dolev, S., Tzachar, N.: Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. Theoretical Computer Science 410, 514–532 (2009), `http://www.sciencedirect.com/science/article/pii/S0304397508007548`, principles of Distributed Systems
8. Gerla, M., Chieh Tsai, J.T.: Multicluster, mobile, multimedia radio network. Journal of Wireless Networks 1, 255–265 (1995)
9. Myoupo, J.F., Cheikhna, A.O., Sow, I.: A randomized clustering of anonymous wireless ad hoc networks with an application to the initialization problem. J. Supercomput. 52(2), 135–148 (2010), `http://dx.doi.org/10.1007/s11227-009-0274-9`
10. Nolte, T., Lynch, N.: Self-stabilization and virtual node layer emulations. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 394–408. Springer, Heidelberg (2007), `http://dl.acm.org/citation.cfm?id=1785110.1785140`