View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Best-effort Group Service in Dynamic Networks

Bertrand Ducourthial, Sofiane Khalfallah, Franck Petit

To cite this version:

Bertrand Ducourthial, Sofiane Khalfallah, Franck Petit. Best-effort Group Service in Dynamic Networks. 2008. hal-00332722v3

HAL Id: hal-00332722

https://hal.archives-ouvertes.fr/hal-00332722v3

Preprint submitted on 10 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Best-effort Group Service in Dynamic Networks*

Bertrand Ducourthial‡ Sofiane Khalfallah‡

Franck Petit†

‡ (1) Université de Technologie de Compiègne
(2) CNRS Heudiasyc UMR6599
Centre de Recherche de Royallieu
B.P. 20529, Compiègne, France

† (1) Université PM. Curie
(2) CNRS LIP6 UMR7606
(3) INRIA REGAL
4, place Jussieu 75005 Paris, France

Abstract

We propose a group membership service for dynamic ad hoc networks. It maintains as long as possible the existing groups and ensures that each group diameter is always smaller than a constant, fixed according to the application using the groups. The proposed protocol is self-stabilizing and works in dynamic distributed systems. Moreover, it ensures a kind of *continuity* in the service offer to the application while the system is converging, except if too strong topology changes happen. Such a *best effort* behavior allows applications to rely on the groups while the stabilization has not been reached, which is very useful in dynamic ad hoc networks.

Keywords: Group maintenance, Best effort, Stabilization, Dynamic network.

1 Introduction

Self-stabilization in dynamic networks A *dynamic* network can be seen as an (*a priori* infinite) sequence of networks over time. In this paper, we focus on dynamic *mobile* networks. Examples of such networks are *Mobile Ad hoc* networks (MANETs) or *Vehicular Ad hoc* networks (VANETs).

Designing applications on top of such networks require dealing with the lack of infrastructure [22, 15]. One idea consists in building virtual structures such as clusters, backbones, or spanning trees. However, when the nodes are moving, the maintenance of such structures may require more control. The dynamic of the network increases the control overhead. Thus, distributed algorithms should require less overall organization of the system in order to remain useful in dynamic networks.

Another paradigm for building distributed protocols in mobile ad hoc networks consists in designing self-stabilizing algorithms [4]. These algorithms have the ability to recover by themselves (*i.e.*, automatically) from an inconsistent state caused by transient failures that may affect a memory or a message. In this context, the topology changes can be considered as transient failures because they lead to an inconsistency in some memories. Indeed, when a node appears or disappears in the network, all its neighbors should update their neighborhood knowledge.

Self-stabilizing algorithms have been intensively studied the two last decades for their ability to tolerate transient faults [9]. However, it is important to notice that such algorithms do not ensure all the time the desirable behavior of the distributed system, especially when faults occur and during a certain period of time following them. In dynamic systems, it becomes illusory to expect an application that continuously ensures the service for which it has been designed. In other words, what we can only expect from the distributed algorithms is to behave as "the best" as possible, the result depending on the dynamic of the network.

*Supported by Région Picardie, proj. APREDY.

In this paper, we propose a new approach in the design of distributed solutions for dynamic environments. We borrow the term "*best-effort*" from the networking community to qualify the algorithms resulting of our approach. Roughly speaking, a best-effort algorithm is a self-stabilizing algorithm that also maintains an extra property, called *continuity*, conditioned by the topology changes.

Continuity aims to improve the output of the distributed protocol during the convergence phase of the algorithm, provided that a topological property is preserved. This means that there is a progression in the successive outputs of the distributed protocols, except if the network dynamic is too high. This is important in a distributed system where the dynamic (that is, the frequent topology changes) can prevent the system to converge to the desirable behavior. Since the output of the protocol will certainly be used before the stabilization, the continuity ensures that third party applications can rely on it instead of waiting. The output will certainly be modified in the future, but without challenging previous ones.

In some aspects, our approach is close to the ones introduced in [17] and in [10]. In [17], the authors introduce the notion of *safe-convergence* which guarantees that the system quickly converges to a safe configuration, and then, it gracefully moves to an optimal configuration without breaking safety. However, the solution in [17] works on a static network. In [10], the authors use the notion of *passage predicate* to define a *superstabilizing* system, *i.e.*, a system which is stabilizing and when it is started from a legitimate state and a single topology change occurs, the passage predicate holds and continues to hold until the protocol reaches a legitimate state. By contrast, the continuity property is intended to be satisfied *before* a legitimate configuration has been reached. It must be satisfied during the stabilization phase, and between two consecutive stabilization phases (convergence phase followed by stability phase).

We illustrate our approach by specifying a new problem, called *Dynamic Group Service* inspired from vehicular ad hoc networks (VANET), an emblematic case of dynamic ad hoc networks. We then design a best effort distributed protocol called `GRP` for solving this problem: we prove that it is self-stabilizing and fulfills a continuity property, allowing applications to use the groups while the convergence may be delayed because of the dynamic of the network.

Dynamic group service Vehicular ad hoc networks currently attract a lot of attention [3]. Many VANET applications require cooperation among close vehicles during a given period: collaborative driving, distributed perception, chats and other infotainment applications. Vehicles that collaborate form a *group*. A group is intended to grow until a limit depending on the application. For instance, the distributed perception should not involve too far vehicles, a chat should be responsive enough, that limits the number of hops, etc. When the group diameter is larger than the bound given by the application, it should be split into several smaller groups. However, a group should not be split if this is not mandatory by the diameter constraint in order to ensure the best duration of service to the application relying on it. Even if another partitioning of the network would have been better (*e.g.*, less groups, no isolated vehicle), it is preferable to maintain the composition of existing groups. It is expected that, thanks to the mobility of the nodes, small groups will eventually succeed in merging. It is then more important to maintain existing groups as long as possible.

Best-effort `GRP` algorithm To solve the *Dynamic Group Service*, we propose a best-effort distributed algorithm called `GRP` (for `GRouP`) designed for unreliable message passing systems. This algorithm stabilizes the views (the local knowledge of the group to which belongs the node) in such a way that all the members of a group will eventually share the same view (in which only the members appear). The groups' diameters are smaller than a fixed applicative constant `Dmax` and neighbor groups merge while the diameter constraint is fulfilled. Moreover, our algorithm admits the following continuity property: no node disappears from a group except if a topology change leads to the violation of the diameter constraint. This allows to the applications requiring the groups (*e.g.*, chat) to

run before the convergence of GRP, that may be delayed because of the dynamic of the network.

To the best of our knowledge, only a few number of papers address the problem of group membership maintenance in the context of self-stabilization. Recently, in [6], the authors propose a self-stabilizing k-clustering algorithm for static networks. In [11], the authors propose a self-stabilizing group communication protocol. It relies on a mobile agent that collects and distributes information during a random walk. This protocol does not allow building groups that strech over at most k hops.

Group communication structures have been proposed in the literature to achieve fault-tolerance in distributed systems [2], by providing for instance replication, virtual synchrony, reliable broadcast, or atomic broadcast (*e.g.*, [21, 14]). Other works deal with the k-clustering or k-dominating set problem, *e.g.*, [1, 5, 8, 16, 17, 18, 20], where nodes in a group are at most at distance k from a *cluster-head* or *dominant node*. The aim of these algorithms is to optimize the partitioning of the network. The group service we propose in this paper is different in the sense that its aim is neither to optimize any partitioning nor to build group centered to some nodes. Instead, it tries to maintain existing groups as long as possible while satisfying a constraint on the diameter, without relying on a specific node (that may move or leave).

Organization In Section 2, we describe the distributed system we consider in this paper. We also state what it means for a protocol to be self-stabilizing and best effort regarding a continuity property conditioned by topology changes. Next, in Section 3, we specify the Dynamic Group Service problem and in Section 4, we describe our GRP algorithm solving it[1]. The proofs are given in Section 5. Finally, we make some concluding remarks in Section 6. By lack of place, some proofs are in appendix.

2 Model

We define the distributed system $\mathcal{S}$ as follows.

System Let V be the set of nodes, spread out in an Euclidean space. The total number of nodes in V is finite but unknown. Each node is equipped with a processor unit (including local memory) and a communication device. A node can move in the Euclidean space. It is either *active* or *inactive*. When it is active, it can compute, send and receive messages by executing a local algorithm. The *distributed protocol* $\mathcal{P}$ is composed of all the local algorithms.

We define the *vicinity* of a node v as the part of the Euclidean space from where a node u can send a message that can be received by v (the vicinity depends on the communication devices, the obstacles, etc.). A node v can receive a message from u if (i) both u and v are active, (ii) u is in the vicinity of v, (iii) u is sending a message, (iv) no other node in the vicinity of v is currently sending a message, and (v) v is not sending a message itself (any active node that is not sending is able to receive).

We assume that on each node the message sending is driven by a *timer*. We admit the following *fair channel* hypothesis: there exists two time constants τ_1 and τ_2 with $\tau_1 \geq \tau_2$ such that, starting from a date t, any node v is able to receive before the date $t + \tau_1$ a message from each node u, providing that u is in the vicinity of v between t and $t + \tau_1$ and attempts to send a message every τ_2 units of time. At any time instant t, there is a *communication link* from u to v if both u and v have the state active (at t), and if u is into the vicinity of v (at t). A communication link is oriented because u could be in the vicinity of v while the converse is false.

[1]. Note that the algorithm has been successfully implemented using the Airplug software suite. The detailed algorithm used for the implementation is available on our website (as long as the software):
`http://www.hds.utc.fr/~ducourth/airplug/doku.php?id=en:dwl:grp:accueil`. Some screenshot movies are also available here:
`http://www.hds.utc.fr/~ducourth/airplug/doku.php?id=en:doc`

We assume the following hypotheses, close to the IEEE 802.11 protocol. (i) The communication channel contains at most one message—*one-message channel.* (ii) If a node u keeps continuously sending a message m, then u eventually sends m—*fair sending.* (iii) If a node u keeps continuously receiving a message, then u eventually receives a message—*fair reception.* (iv) If a node u has continuously an action a to execute, then u executes a in finite time—*fair activation.*

A *configuration* c of $\mathcal{S}$ is the union of states of memories of all the processors and the contents of all the communication links. An empty communication link is denoted in the configuration by a link that contains an empty set of messages. By the way, there is a single topology per configuration. Let $\mathcal{C}$ be the set of configurations. An *execution* of a distributed protocol $\mathcal{P}$ over $\mathcal{S}$ is a sequence of configurations $c_0, c_1, \ldots$ of $\mathcal{S}$ so that $\forall i \leq 0$, c_i moves to c_{i+1} by changing the memory of at least one process, including its message buffers (*i.e.*, by sending or receiving messages).

We denote by G^{c_i} the topology of $\mathcal{S}$ during the configuration c_i. In a *static* system $\mathcal{S}$, we have $G^{c_i} = G^{c_0}$ in every execution $c_0, c_1, c_2, \ldots$. Otherwise, the system $\mathcal{S}$ is said to be *dynamic.*

Self-Stabilization Let $\mathcal{X}$ be a set. Then $x \vdash \Pi$ means that an element $x \in \mathcal{X}$ satisfies the predicate Π defined on the set $\mathcal{X}$ and $X \vdash \Pi$ with $X \subset \mathcal{X}$ means that any $x \in X$ satisfies $x \vdash \Pi$. We define a special predicate true as follows: $\forall x \in \mathcal{X}$, $x \vdash \mathsf{true}$. Let Π_1 and Π_2 be two predicates defined on the set of configurations $\mathcal{C}$ of the system $\mathcal{S}$. Π_2 is an *attractor* for Π_1 if and only if the following condition is true: for any configuration $c_1 \vdash \Pi_1$ and for any execution $e = c_1, c_2, \ldots$, there exists $i \geq 1$ such that for any $j \geq i$, $c_j \vdash \Pi_2$.

Define a *specification* of a task as the predicate Π on the set $\mathcal{C}$ of configurations of system $\mathcal{S}$. A protocol $\mathcal{P}$ is self-stabilizing for Π if and only if there exists a predicate $\mathcal{L}_{\mathcal{P}}$ (called the legitimacy predicate) defined on $\mathcal{C}$ such that the following conditions hold:
1. For any configuration $c_1 \vdash \mathcal{L}_{\mathcal{P}}$, and for any execution $e = c_1, c_2, \ldots$, we have $e \vdash \Pi$ (correctness).
2. Π is an attractor for true (closure and convergence).

Best effort requirement We denote by Π_T a *topological* predicate defined on the pairs of successive configurations in an execution. Such a predicate is intended to be false when an "important topology" change happens. We denote by Π_C a *continuity* predicate defined on the pairs of successive configurations in an execution. Such a predicate is intended to be false when the quality of the outputs produced by protocol $\mathcal{P}$ in the two successive configurations decreases.

The protocol $\mathcal{P}$ offers a best effort continuity of services if $\Pi_T \Rightarrow \Pi_C$.

3 Dynamic Group Service Problem

The Dynamic Group Service protocol is inspired from applications requirements in Vehicular Ad Hoc networks (VANET), such as collaborative perception or infotainment applications.

Informal specification On each node v, a variable $\texttt{view}_v$ gives the composition of the group to which v belongs. This will be used by the applications. The agreement property says that all nodes in group of v agree on the composition of the group. The safety property says that the diameter of each group is smaller than a constant `Dmax`. The maximality property says that small groups merge to form larger groups.

To deal with the dynamic of the network, the algorithm should be able to satisfy these three properties in finite time after the last failure or topology change (self-stabilization). To allow the applications to run while the convergence has not been reached, the algorithm should ensure a best effort requirement: if the distance between the members of a group remains smaller than `Dmax` (topological property), then no node will leave the group (continuity property). This is important because the convergence may be delayed because of the dynamic of the network.

Formal specification Let $G(V,E)$ be a graph. Let $d(u,v)$ be the *distance* between u and v (length of the shortest path from u to v in G). A *subgraph* $H(V_H,E_H)$ is defined as follows: $V_H \subseteq V$ and $\forall (u,v) \in E,\ (u \in V_H \text{ and } v \in V_H) \Rightarrow (u,v) \in E_H$. Two subgraphs $H_1(V_1,E_1)$ and $H_2(V_2,E_2)$ of a graph G are said *distinct* if $V_1 \cap V_2 = \emptyset$. Let $X \subseteq V$ be a set of nodes. We denote by $d_X(u,v)$ the distance between u and v in the subgraph $H(X,E_H)$, that is, the length of the shortest path from u to v with only edges of E_H. If such a path does not exists, then $d_X(u,v) = +\infty$.

Given a graph G, the problem considered in this paper consists in designing a distributed protocol that provides a partition of G into disjoint subgraphs called *groups* that satisfies constraints described below. Denote by $\texttt{view}_v^c$ the knowledge of v about its group in configuration c (output on node v).

Let Π_A be the predicate defined on the configurations and called *agreement property*: $\Pi_A(c)$ holds if and only if there exists a partition of disjoint subgraphs $H_1(V_1,E_1)$, $H_2(V_2,E_2)$, ..., $H_i(V_i,E_i)$, ... of $G(V,E)$ such that for every nodes $u,v \in V$, $(u \in V_i \text{ and } v \in V_i) \Leftrightarrow \texttt{view}_u^c = \texttt{view}_v^c = V_i$.

Let Ω_v^c be the *group* of v in configuration c, defined by: (i) $\Omega_v^c = \texttt{view}_v^c$ if $v \in \texttt{view}_v^c$ and $\forall u \in \texttt{view}_v^c, \texttt{view}_v^c = \texttt{view}_u^c$, (ii) $\Omega_v^c = \{v\}$ otherwise. Note that given any configuration c, if $\Pi_A(c)$ holds, $\{\Omega_v^c, v \in V\}$ defines a partition of G into disjoint subgraphs of G, *i.e.*, there exists a partition of disjoint subgraphs $H_1(V_1,E_1)$, $H_2(V_2,E_2)$, ..., $H_i(V_i,E_i)$, ... such that $\forall v \in V_i,\ \Omega_v^c = V_i$ for every subgraph H_i.

Let $\texttt{Dmax}$ be an integer representing the maximal admissible distance between two nodes belonging to the same group. Let Π_S be the predicate defined on the configurations and called *safety property*: $\Pi_S(c)$ holds if each group is connected and its diameter is smaller than $\texttt{Dmax}$. More formally $\Pi_S(c) \equiv \forall v \in V$, $\max_{x,y \in \Omega_v^c} d_{\Omega_v^c}(x,y) \leq \texttt{Dmax}$.

Let Π_M be the predicate defined on the configurations and called *maximality property*: $\Pi_M(c)$ holds if by merging two existing groups, we cannot obtain a partition satisfying the safety property. More formally $\Pi_M(c) \equiv \forall u,v \in V$ with $\Omega_u^c \neq \Omega_v^c$, $\exists x,y \in \Omega_u^c \cup \Omega_v^c$, $d_{\Omega_u^c \cup \Omega_v^c}(x,y) > \texttt{Dmax}$.

The problem considered in this paper is to design a self-stabilizing protocol regarding predicates $\Pi_A \wedge \Pi_S \wedge \Pi_M$: after the last failure or topology change, the algorithm converges in finite time to a behavior where Π_A, Π_S, and Π_M are fulfilled.

Note that the above requirement is suitable for fixed topologies only. The following predicate deals with dynamic system, *i.e.*, with topological changes. Let $G^c(V^c,E^c)$ be the graph modeling the topology of the system at configuration c. We introduce the following notation: d^c refers to the distance in the graph G^c, and $d_X^c(u,v)$ denotes the distance between u and v in G^c by considering only edges of the subgraph $H(X,E_H)$ of G^c. Define the *topological property* as the predicate Π_T defined on any couple of two successive configurations c_i, c_{i+1} of an execution e as follows: $\Pi_T(c_i,c_{i+1})$ holds if, for any pair of nodes belonging to the same group in c_i, the distance between them will still be smaller than $\texttt{Dmax}$ in c_{i+1}. In other words, if a topology change occurred between c_i and c_{i+1}, it has preserved the maximal distance condition. More formally, $\Pi_T(c_i,c_{i+1}) \equiv \forall v \in V$, $\max_{x,y \in \Omega_v^{c_i}} d_{\Omega_v^{c_i}}^{c_{i+1}}(x,y) \leq \texttt{Dmax}$.

Finally, we are looking for protocols attempting to preserve a group partition when a topology change occurs. Let Π_C be the predicate defined on the couples of successive configurations and called *continuity property*: $\Pi_C(c_i,c_{i+1})$ holds if in any group, no node disappears. In other words, an application can work with the given view because it defines a group in which no node will disappear. More formally, $\Pi_C(c_i,c_{i+1}) \equiv \forall v \in V$, $\Omega_v^{c_i} \subset \Omega_v^{c_{i+1}}$. Obviously, if the dynamic of the network is too large, such a property cannot be satisfied. We then introduce the best effort requirement: $\Pi_T \Rightarrow \Pi_C$.

4 GRP distributed protocol

The GRP distributed protocol is designed for solving the Distributed Group Service problem in an unreliable message passing system.

4.1 Principle of the GRP distributed protocol

For each node v, the candidates to form a group are neighbors up to distance Dmax. Each node v periodically echanges messages with its neighbors and maintains a list of nodes being at distance at most Dmax. Each sent message sent by v contains the list of v. The list of v contains nodes at distance at most Dmax that are in the group or candidates to join the group.

Our mechenism needs to take in account symmetric links only, *i.e.*, links between pairs of nodes u and v so that if v is considered by u as a neighbor, then u (resp. v) is considered as a neighbor by v. In order to implement this, we use marks. Each node proceeds as follows: if v receives a list from u that does not contain itself, then it adds $\underline{u}$ in its list (which will be sent to the neighbors at the next timer expiration). To the converse, when v receives a list from u that contains either v or $\underline{v}$, then it adds u in its list. Marked nodes are not propagated farther than the neighborhood.

Malformed lists are rejected (such as lists larger than Dmax). Moreover, when a node v receives a list from u which is too long compared to its current list, it rejects it to avoid any split of its current group. In this case, v adds $\underline{\underline{u}}$ in its list, meaning that u and v cannot belong to the same group. To the converse, if the received list is not too long, it is merged with the current list, meaning that u enters to the group of v. Symmetrically, u will accept v in its group.

Several nodes may be accepted concurrently by distant members of a given group. In some cases, a too large group may be obtained. Then one of the new members must leave the group (instead of splitting the existing group). To avoid any inopportune change in the views (which are used by the applications), a new member enters in the view of a node only after the end of its *quarantine* period. This allows guaranteeing that its arrival has been approved by all the members (no conflicts). A node arrival is propagated to the group's members in $O(\texttt{Dmax})$; this defines the quarantine period duration.

When it is necessary to chose which node has to leave the group (to fulfill the diameter constraint), the choice is done using a *priority* computed by Function *texttpr*. Priorities are totally ordered; if $\texttt{pr}(u) < \texttt{pr}(v)$, then u has the priority. A powerful implementation of priorities is the oldness of nodes in the groups: the priority of a node is incremented by a logical clock [19], except if it belongs to a group (of more than one node) in which case the priority remains stable. The last entered nodes in a group have then less priority than the nodes entered before them.

Priorities on the nodes allow to easily define priorities on the groups by taking the smallest priority of the members. Priorities on the groups allows to ensure the merging of neighbor groups (and the maximality property Π_M) in particular cases (loop of groups willing to merge).

4.2 Building the lists

In the sequel, a node v is an *ancestor* of node u if a path exists from u to v. The messages sent to the neighbors contain *ordered list of ancestors' sets*. The *ordered list of ancestors' sets* of a node v is defined by: $\left(a_v^0,\ a_v^1,\ \ldots,\ a_v^p\right)$ where any node $x \in a_v^i$ satisfies $d(x,v) = i$ ($a_v^0 = \{v\}$) and p is the distance of the farthest ancestor of v.

Computations are done using the r-operator ant [7, 13, 12]. Let $\mathbb{S}$ be the set of lists of vertices' sets. For instance, if a, b, c, d, e are vertices, $(\{d\}, \{b\}, \{a, c\})$ and $(\{c\}, \{a, e\}, \{b\})$ belong to $\mathbb{S}$. Let $\oplus$ be the operator defined on $\mathbb{S}$ that merges two lists while deleting needless or repetitive information (a node appears only one time in a list of ancestors' sets). For instance:
$(\{d\}, \{b\}, \{a, c\}) \oplus (\{c\}, \{a, e\}, \{b\}) =$
$(\{d, c\}, \{b, a, e\}, \{a, c, b\}) = (\{d, c\}, \{b, a, e\})$.

Finally, let r be the endomorphism of $\mathbb{S}$ that inserts an empty set at the beginning of a list. For instance:
$r(\{d\}, \{b\}, \{a, c\}) = (\emptyset, \{d\}, \{b\}, \{a, c\})$.

We then define the operator ant by: $\text{ant}(l_1, l_2) = l_1 \oplus r(l_2)$, where l_1 and l_2 are lists belonging to $\mathbb{S}$. This is a strictly idempotent r-operator [12] inducing a partial order relation. It leads to self-

stabilizing static tasks (building the complete ordered lists of ancestor sets) in the register model [13]. Since our wireless communication model admits bounded links, these results can be extended to this model. (Refer to the discussion related to r-operators in wireless networks in [7].)

4.3 GRP algorithm

Each node v computes its output ($\texttt{list}_v$, $\texttt{view}_v$ and the priorities) when its timer T_c expires. It broadcasts its output in the neighborhood when the timer T_s expires ($T_s \leq T_c$). All messages received from the neighborhood are collected on v in $\texttt{msgSet}_v$. If a neighbor sends more than one message before the timer expiration, only the last received is kept. After computation, the variable $\texttt{msgSet}_v$ is reset in order to detect when a neighbor leaves.

Algorithm GRP, node v

1 Upon reception of a message msg sent by a node u:
2 update message of u in $\texttt{msgSet}_v$

3 Upon T_c timer expiration:
4 compute()
5 reset $\texttt{msgSet}_v$
6 restart timer T_c with duration τ_1

7 Upon T_s timer expiration:
8 **send(** $\texttt{list}_v$ with priorities **)** to the neighbors
9 restart timer T_s with duration τ_2

A computation (in procedure `compute`, below) consists in building the ordered list of ancestor' sets as well as the view. The list is sent to the neighbors to be used in their ant computation. The view is the output of the protocol used by the applications (*e.g.*, chat, collaborative perception...) which requested the GRP algorithm, and which determined the diameter constraint `Dmax` (fixed during all the execution).

First, the incoming lists are checked. Line 3, when the list sent by u and received by v does not contain v, is malformed or is too long[2], it is replaced by $(\underline{u})$. When u receives the list of v containing $\underline{u}$, it accepts the list of v and sends a list containing v. Thanks to this triple handshake, the link has been detected as symmetric (by the way, asymmetric link information are not propagated).

Line 6, if the received list is too long, the sender u is marked as incompatible $(\underline{\underline{u}})$. Roughly speaking, a list received by a node u from another node v is compatible if, by combining its list with the one of v, u does not increase the diameter of its group beyond `Dmax`. In order to reach this goal, it is enough to test if the sum of the lengths of both lists is less than or equal to $\texttt{Dmax} + 1$. But, such simple test would avoid merging two groups by taking advantage of short cuts between both groups. In other words, this would ignore the knowledge that nodes of a group have on nodes belonging to the other group. The technical condition used in Function `compatibleList()` deals with such an optimization.

Then a first computation is performed using the ant operator. Thanks to the `goodList` test, the sizes of the incoming lists are smaller than $\texttt{Dmax} + 1$. However, the computed list could reach the size of $\texttt{Dmax} + 2$ while the maximum is $\texttt{Dmax} + 1$ (the ant operation increases by one the list sizes). In this case, a choice has to be done between either the local node v or the farthest nodes in the received lists. This choice is done by using priorities, Line 16. If nodes belong to the same group, node priorities are compared. If nodes are not in the same group, this is a group merging and group priorities are compared (to avoid loops of groups willing to merge). If the local node v has not the priority on the too far node w the lists in which w appears are ignored (Line 19). At the opposite side of the group, node w keeps the list containing v but the end of its ordered list of ancestor's sets will be truncated

[2] $s(\texttt{list})$ returns the number of elements in `list`; $\texttt{list}.i$ returns the ith element of `list`, starting from 0.

(meaning that v and w will not belong to the same group). Indeed, after the too far nodes have been all examined, the list of ancestors is computed again (Lines 24-27) and is truncated (Line 28) in order to delete the too far nodes (these remaining too fare nodes have less priority than v).

In order to not include a node in a view while it could be rejected later, a *quarantine mechanism* is used. The quarantine period of a node willing to enter in a group is equal to `Dmax` timers. Each time a computation is done (and then the new node progresses in the group), its quarantine period decreases. Since the group diameter is less than or equal to `Dmax`, any conflict would have been detected before the new node enters into a view. Moreover, if a member of the group accepts the new node, then all the members will accept it.

Finally, the priority is updated. When using oldness in the group, the priority is increased if the node is not in a group. If the node is in a group, the priority remains stable. The group priority is the smallest priority of its members.

The procedure `compute()` is given below (a complete implementation with the detailed algorithm is available on-line, see reference in Footnote 3 page 3).

Procedure compute() on node v

```
    ▷ Checking the received lists
1   for all list_u in msgSet do
2      delete marked nodes except v̲ in list_u          ▷ Marked nodes are only useful between neighbors.
3      if ¬ goodList(list_u) then                      ▷ List of u cannot be used;
4         replace list_u by (u̲) in msgSet              ▷ this list is ignored but the sender is kept.
5      end if                                          ▷ Now, incoming lists cannot be larger than Dmax.
6      if u ∉ view_v and ¬ compatibleList(list_u) then ▷ u is new, but its list cannot be accepted;
7         replace list_u by (u̲̲) in msgSet              ▷ u is denoted as an incompatible neighbor
8      end if
9   end for
    ▷ Computing the list of ancestors' sets of v.
10  list_v ← (v)
11  for all list_u ∈ msgSet do
12     list_v ← ant(list_v, list_u)                    ▷ Computation using the ant r-operator.
13  end for
    ▷ Removal of incoming lists containing too far nodes (after ant computation, list_v cannot be larger than Dmax+1)
14  if s(list_v) = Dmax + 2 then                       ▷ The list is too long.
15     for all w at position Dmax + 1 in list_v do     ▷ Scanning too far nodes.
16        if w has the priority compared to v then     ▷ Far node w has the priority.
17           for all list_u ∈ msgSet do                ▷ Looking for lists that provided w;
18              if w is at position Dmax then          ▷ they contain w in their last place.
19                 replace list_u by (u̲̲) in msgSet      ▷ The neighbor that provided w is ignored.
20              end if
21           end for
22        end if
23     end for
       ▷ Computing list_v again, without the incoming lists that contained too far nodes with priority.
24     list_v ← (v)
25     for all list_u in msgSet do
26        list_v ← ant(list_v, list_u)
27     end for
28     keeping up to Dmax + 1 first elements in list_v  ▷ Deleted too far nodes have not the priority.
29  end if
30  Update quarantines: quarantine of new nodes is Dmax, non null quarantine of others decreases by 1
31  view_v ← non marked nodes in list_v with null quarantine
32  Update priorities: priority of nodes increase only when they are not in a group
```

Function goodList(list)

```
1   if v or v̲ are in list.1 and s(list) ≤ Dmax + 1 and ∅ ∉ list then
2      return true
3   else
```

```
4        return false

Function compatibleList(list)
1    if s(list_v) + s(list) ≤ Dmax + 1 or
       ∃i ∈ {0, ..., s(list_v)}, list_v.i ⊆ list.1 ∧ min(s(list_v) + s(list) + 1 − i, s(list) + 1 + i/2) ≤ Dmax
2    return true                                                    ▷ Refer to Proposition 13.
3  else
4    return false
```

5 Proofs

We first focus on the self-stabilizing property of our algorithm. We show that assuming a fixed topology, the system converges in finite time to an execution satisfying the statements in Section 3, *i.e.*, $\Pi_S \wedge \Pi_A \wedge \Pi_M$ is an attractor. Next, we prove that, assuming topological changes preserving the maximal distance condition over the groups, then continuity is preserved, *i.e.*, $\Pi_T \Rightarrow \Pi_C$.

5.1 Stabilization

In this section, we prove that our protocol is self-stabilizing by showing that Π_S and Π_A and Π_M are attractors—Propositions 8, 7 and 12, respectively.

We begin by showing that eventually lists will become correct (Propositions 1 and 2). We first prove that any execution cannot remain infinitely with configurations having lists larger than $\texttt{Dmax}$. We denote by $e_{\texttt{Dmax}}$ the suffix of an execution e such that, for any configuration $c \in e_{\texttt{Dmax}}$, for any node $v \in V$, the size of $\texttt{list}_v$ is smaller than or equal to $\texttt{Dmax} + 1$.

Proposition 1 (Dmax) *On a fixed topology, any execution e reaches in finite time a suffix* $e_{\texttt{Dmax}}$.

Proof. Starting from configuration c_1, the system will reach in finite time a configuration in which every node has computed its list after expiration of its timer. After such a computation, the size of the lists is bounded by $\texttt{Dmax} + 1$ (because it is truncated at the $\texttt{Dmax} + 1$ position, Line 28). □

Starting from this proposition, we now prove that any execution cannot remain infinitely with configurations having a non existing node in a list. We denote by $e_{\texttt{exist}}$ the suffix of an execution e such that, for any configuration $c \in e_{\texttt{exist}}$, for any node $v \in V$, every node $u \in \texttt{list}_v^c$ satisfies $u \in V$.

Proposition 2 (Exist) *On a fixed topology, any execution e reaches in finite time a suffix* $e_{\texttt{exist}}$.

Proof. Let $c \in e_{\texttt{Dmax}}$ be a configuration (Proposition 1). Let u be a node label such that $u \notin V$ and denote by U_k^c the set of nodes having u in their list at position k in configuration c. Consider the function $\phi(c)$ defined by $\phi(c) = \min\{k \in \mathbb{N}, U_k^c \neq \emptyset\}$ and $\phi(c) = \infty$ if $\forall k \in \mathbb{N}, U_k^c = \emptyset$. We prove that ϕ is continuously growing along the execution to be eventually equal to infinity forever.

Consider a node v in $U_{\phi(c)}^c$: v contains u at position $\phi(c)$ in its computed list and no node in configuration c contains u at a smaller position in its computed list. Until the next expiration of its timer, v cannot receive a list containing u in a smaller position than $\phi(c)$. Hence, the system will reach in finite time a configuration in which the node v has computed a new list that does not contain u at a position smaller than $\phi(c) + 1$. After a timer (fair channel Hypothesis), the system reaches in finite time a configuration in which the neighbors of v have received this list.

After finite time, any node $v \in U_{\phi(c)}^c$ will do the same. The system then reaches in finite time after configuration c a configuration c' in which $U_{\phi(c)}^{c'}$ is empty, meaning that $\phi(c) < \phi(c')$.

By iteration, ϕ is growing along the execution. Since the size of the lists is bounded by $\texttt{Dmax} + 1$ (Proposition 1), there exists a configuration c'' reached in finite time after c in which $\phi(c'') = \infty$, meaning that u does not appear anymore in the computed lists of the nodes forever. □

Next, we establish the connection between marked nodes in the algorithms and subgraphs (Propositions 3, 4, 5 and 6). We call *double-marked edge* an edge (u, v) such that either u double-marks v or v double-marks u (denoted by $\underline{\underline{u}}$ in the algorithm). The following proposition is a consequence of the double-marked edge technique. A node v double-marks its neighbor u only if the list sent by u cannot be accepted by v (Lines 7 and 19). In this case, node v will ignore the list sent by u. Reciprocally, if u has been double-marked by v, u will detect an asymmetric link (u does not appear in the list it received after Line 2) and only the identity of v will be kept by u, the rest of the list of v will be ignored (Line 4).

Proposition 3 (No Propagation) *Let u and v be two vertices of G and suppose that, in any execution e, there exists a configuration c_e from which any path from u to v in G contains a double-marked edge. Then u will eventually disappear from $\texttt{list}_v^c$ and v will eventually disappear from $\texttt{list}_u^c$.*

The following proposition is a consequence of the *ant* computation (see Section 4.2). It propagates nodes identities (providing there is no edge-marking technique for limiting it) [13, 7].

Proposition 4 (Propagation) *Let u and v be two vertices of G and suppose that, in any execution e, there exists a configuration c_e from which there exists a path from u to v in G without double-marked edge. Then $\texttt{list}_v^c$ will eventually contain u and $\texttt{list}_u^c$ will eventually contain v.*

Proposition 5 (Double-marked edge) *Suppose that $d(u, v) > \texttt{Dmax}$. Then any execution admits a suffix $e_{\texttt{edge}}$ such that, for any configuration $c \in e_{\texttt{edge}}$, there is a double-marked edge on any path from u to v.*

Proof. Let v and w two nodes of G such that $d(v, w) = \texttt{Dmax} + 1$. Without loss of generality, we suppose that $pr(w) < pr(v)$. Suppose that there exists a path from v to w that does not contain any double-marked edge. By Proposition 4, there exists a neighbor u of v such that u sends to v a list containing w. The size of this list is larger than $\texttt{Dmax}$. There are two cases:
(i) $u \notin \texttt{view}_v$. In this case, $\texttt{list}_u$ is replaced by $(\underline{\underline{u}})$.
(ii) $u \in \texttt{view}_v$. In this case, v computes a list using the one sent by u. Since $d(u, v) > \texttt{Dmax}$, the resulting list is too long. Since $pr(w) < pr(v)$, the computation will be done again without the list provided by u, which will be replaced by $(\underline{\underline{u}})$.

In the two cases, u is double-marked by v. Hence, any path from u to v will eventually contains a double-marked edge. □

Let denote by $H_v^c(V_{H_v}, E_{H_v})$ the subgraph of $G(V, E)$ defined in the configuration c by: for any node u in V_{H_v}, $v \in \texttt{list}_u^c$. Such a subgraph is composed of vertices containing v in their list. We prove that eventually H_u and H_v are distinct when $d(u, v) > \texttt{Dmax}$.

Proposition 6 (Subgraphs) *Suppose that $d(u, v) > \texttt{Dmax}$. Then any execution admits a suffix $e_{\texttt{subgraph}}$ such that, for any configuration $c \in e_{\texttt{subgraph}}$, H_u and H_v are distinct subgraphs.*

Proof. By Proposition 5, there exists a suffix s_1 such that any path from u to v contains a double-marked edge. By Proposition 3, there exists a suffix s_2 included in s_1 such that for any configuration c in this suffix, $u \notin \texttt{list}_v^c$ and $v \notin \texttt{list}_u^c$. Then $u \notin H_v$ and $v \notin H_u$.

Let consider a node w such that $w \in H_v$ and $w \in H_u$. Then there exists at least one path from u to v containing w. The length of such a path is larger than $\texttt{Dmax}$. Then, by Proposition 5, it admits a double-marked edge, either on the subpath from u to w or from the subpath from w to v.

Now, let consider all the paths from u to v containing w; they all contain a double-marked edge. Suppose that for one path P_1, this double-marked edge is between w and v and for a second path P_2, it is between u and w. Then, by considering edges of P_1 from u to w and edges of P_2 from w to v, we obtain a path from u to v without any double-marked edge, which is a contradiction. Then, all paths from u to v containing w admit a double-marked edge, and this edge is always between u and w or always between w and v. Thus, w cannot belong to both H_u and H_v, meaning that there is no node w such that $w \in H_u$ and $w \in H_v$.

Hence, any execution reaches a suffix such that, for any configuration c in this suffix, H_u^c and H_v^c are distinct. □

The preceding propositions give the Agreement. Consider any execution $e_{\texttt{subgraphs}}$. Denote by $e_{\texttt{agree}}$ the suffix of an execution e such that $\Pi_A(c)$ holds for any configuration $c \in e_{\texttt{agree}}$, that is $V_{H_v} = \texttt{view}_w^c$ for any $w \in H_v$. The following proposition is given by Propositions 6, 4 and 3.

Proposition 7 (Agreement) *On a fixed topology, any execution e reaches in finite time a suffix e_{agree}.*

Proof. By Proposition 6, for any execution, there exists a suffix such that, for any nodes u and v in G, if $d(u,v) > \texttt{Dmax}$, then the subgraphs H_u and H_v are distinct. Consider now two nodes w and v such that w belongs to H_v

By Proposition 4, for any execution, there exists a suffix such that, for any configuration c in this suffix, the identities of H_v will be in $\texttt{list}_w^c$.

By Proposition 3, for any execution, there exists a suffix such that, for any configuration c in this suffix, the $\texttt{list}_w^c$ contains only vertices of H_v.

After the end of the quarantine period, all the nodes in $\texttt{list}_w$ belong to $\texttt{view}_w$. Then the system reaches a suffix in which all the nodes of H_v and only these nodes appear in $\texttt{view}_w$, for any vertex $w \in H_v$. Hence, $\texttt{view}_v^c = \texttt{view}_w^c = \Omega_v^c$. This gives Π_A. □

Now we have the agreement, there is a connection between subgraphs and groups. We then prove the Safety. Consider any execution $e_{\texttt{agree}}$. Denote by $e_{\texttt{safe}}$ the suffix of an execution e such that $\Pi_S(c)$ holds for any configuration $c \in e_{\texttt{safe}}$. The following proposition is a consequence of Prop. 6.

Proposition 8 (Safety) *On a fixed topology, any execution e reaches in finite time a suffix e_{safe}.*

Proof. By Proposition 6, for any execution and any nodes u and v in G satisfying $d(u,v) > \texttt{Dmax}$, the subgraphs H_u and H_v will eventually be distinct. Hence, for any execution, there exists a suffix $e_{\texttt{safe}}$ such that, for any configuration $c \in e_{\texttt{safe}}$, for any vertex v in G, $\text{Diam}(H_v^c) \leq \texttt{Dmax}$.

Then, by Proposition 7, we have $\max_{x,y \in \Omega_v^c} d_{\Omega_v^c}(x,y) \leq \texttt{Dmax}$. This gives Π_S. □

We consider any execution $e_{\texttt{agree}}$. In order to prove the maximality property, we introduce the following definitions. An edge (u,v) is *internal* in a given configuration c if $\Omega_u^c = \Omega_v^c$. In the converse case ($\Omega_u^c \neq \Omega_v^c$), it is *external*. An external edge involves double-marked nodes and it is then not propagated by the algorithm (marked nodes are deleted, see line 2 in Procedure `compute()`). We denote by *nee* (resp. *ndg*) the function defined on $\mathcal{C}$ that returns the number of external edges in a given configuration (resp. the number of distinct groups in configuration c: $ndg(c) = |\{\Omega_v^c, v \in V\}|$.

Proposition 9 (Nee) *If nee is decreasing along a suffix e_s of an execution e, ndg is also decreasing along e_s.*

Proof. Let (u,v) be an external edge in a configuration c_i and assume that it is an internal edge in configuration c_{i+1}. This means that $\Omega_u^{c_i} \neq \Omega_v^{c_i}$ and $\Omega_u^{c_{i+1}} = \Omega_v^{c_{i+1}}$. Hence $nee(c_i) > nee(c_{i+1}) \Rightarrow ndg(c_i) > ndg(c_{i+1})$. □

We prove that any execution reaches in finite time a suffix in which the function *nee* does not increase. We denote by $e_{\texttt{notincr}}$ such a suffix: $\forall c_i, c_{i+1} \in e_{\texttt{notincr}}$, $nee(c_{i+1}) \leq nee(c_i)$.

Proposition 10 (Not incr.) *On a fixed topology, any execution e reaches in finite time a suffix* $e_{\texttt{notincr}}$.

Proof. Let $c \in e_{\texttt{agree}}$ be a configuration (Proposition 7). Let (u, v) be an internal edge in configuration c. Then we have $\Omega_u^c = \Omega_v^c$ and u is in $\texttt{list}_v^c$. In order (u, v) becomes an external edge, one of its extremity (say v) would have double-marked the other (in Procedure `compute()`). But this cannot happen after the `goodList` test (line 3) because $c \in e_{\texttt{subgraphs}}$. This cannot happen after the `compatibleList` test (line 6) because u is in already in $\texttt{view}_v^c$. □

Now, we prove that any execution reaches in finite time a suffix in which the function *nee* is decreasing while Π_M is not true. We denote by $e_{\texttt{decr}}$ such a suffix: $\forall c_i \in e_{\texttt{decr}}$, $\Pi_M(c_i) \vee \exists c_j \in e_{\texttt{decr}}$, $i < j$ and $nee(c_i) > nee(c_j)$.

Proposition 11 (Decreasing) *On a fixed topology, any execution e reaches in finite time a suffix* e_{decr}.

Proof. Let $c \in e_{\texttt{notincr}}$ be a configuration (Proposition 10). Starting from such a configuration, the *nee* function cannot increase. Suppose that Π_M is not true in c. Then, by definition of Π_M, there exists two neighbors nodes x and y with different views that could merge their groups without breaking Π_S. By fair channel hypothesis, a timer later the system reaches a configuration c' in which x (resp. y) has received the list sent by y (resp. x).

Without loss of generality, suppose that Ω_x has the smallest priority among all the subgraphs that can merge, and Ω_y has the smallest priority among all the groups that can merge with Ω_x.

During the `compute()` Procedure on x and y, the `goodList` tests are true because $c' \in e_{\texttt{notincr}}$ and then $c' \in e_{\texttt{safe}}$. The `compatibleList` test is true on both x and y because they cannot have change their list since configuration c. Hence we obtain: $x \in \texttt{list}_y$ and $y \in \texttt{list}_x$.

Since Ω_y has the smallest priority among the neighbors of Ω_x, no member of Ω_x can receive a message from a group with a smallest priority. Therefore x will never receive and then will never send to y a list with a too far node with a smallest priority than y one's. Hence y will never double-mark x and x will remain in the list of y.

Similarly, since Ω_x has the smallest priority among the groups that can merge, no member of Ω_y can receive a message from a group with a smallest priority. Therefore y will never receive and then will never send to x a list with a too far node with a smallest priority than x one's. Hence x will never double-mark y and y will remain in the list of x.

After `Dmax` timer, the list of y (resp. x) has reached any $u \in \Omega_x$ (resp. Ω_y) thanks to the fair channel Hypothesis. Moreover the quarantine of these new members reaches 0 and they are now included in $\texttt{view}_u$. Thus, the edge (x, y) becomes an internal edge.

Hence, starting from configuration c with $\neg\Pi_M(c)$, the system reaches in finite time a configuration c'' with $nee(c) > nee(c'')$. □

The following proposition is given by Propositions 9, 10 and 11; it shows that any execution reaches in finite time a suffix in which Π_M is true. We denote by $e_{\texttt{max}}$ such a suffix.

Proposition 12 (Maximality) *On a fixed topology, any execution e reaches in finite time a suffix* e_{max}.

Proof. By Prop. 10, the execution reaches a suffix $e_{\texttt{notincr}}$ such that the *nee* function will no more increase. By Prop. 11, the execution reaches a suffix $e_{\texttt{decr}}$ such that the *nee* function decreases while Π_M is not true. Hence, while Π_M is false, the number of external edges will eventually decrease. By Prop. 9, this means that the number of subgraphs will eventually decrease while Π_M is false. Since the graph is finite, the number of subgraphs cannot decrease infinitely and Π_M will eventually become true. □

5.2 Best-effort requirement

We now consider the dynamic of the network. We show that if the continuity property is violated into a group, then there exists a pair of nodes belonging to that group such that the distance between them is larger that `Dmax`. The following technical proposition justifies the compatibleList test.

Proposition 13 (Compatible lists) *Let v be a node having the list $(a_v^0, a_v^1, \ldots, a_v^p)$, $(p \geq 0)$ and assume that its neighbor w sends the list $(a_w^0, a_w^1, \ldots, a_w^q)$, $(q \geq 0)$. Then, the diameter of the group of v after v accepts w remains smaller than or equal to* `Dmax` *if and only if there exists $i \in \{0, \ldots, p\}$ such that w is neighbor of all the nodes belonging to a_v^i and either $p - i + 1 + q \leq$* `Dmax` *or $i/2 + q + 1 \leq$* `Dmax`*.*

Proof. Let $c \in e_{\texttt{safe}}$ be a configuration (Proposition 8). Let w be the first node of Ω_w^c for which the list of ancestor's sets is received by v. Then, the only external edges between Ω_v^c and Ω_w^c known by v are those joining w (external edges are not propagated). Hence, without loss of generality, assume that only these external edges exist between the groups.

($\Rightarrow$) Assume that the conditions are fulfilled. Let $u \in a_v^k$ and $u' \in a_w^l$ be two nodes in the lists of v and w respectively. There exists at most two families of shortest paths from u to u', depending on the external edge used to reach w. Let P_1 be a path that includes the edge (v, w). It starts from u and joins v by k edges in the group of v, joins w by the edge (u, v) and then reaches u' by l edges in the group of u. Let P_2 be a path from the second family. It starts from u and joins a node $v' \in a_v^i$ by $|k - i|$ internal edges in the group of v, then joins w by the edge (v', w) and then reaches u' by l internal edges in the group of u.

The length of P_1 is bounded by $k + 1 + q$. But since P_1 is a shortest path, it is shorter to reach u' from u by joining a node of a_v^0 (*i.e.*, v) than by joining a node of a_v^i (such as v'). Hence we have $k \leq i/2$ and the length of P_1 is bounded $i/2 + 1 + q$, which is smaller than `Dmax` by hypothesis. The length of P_2 is bounded by $p - i + 1 + q$, which is also smaller than `Dmax` by assumption.

Hence, for any node u and u' belonging to the group of v and w respectively, there exists a path from u to u' with less than `Dmax` edges. The list of w is then compatible with the list of v, and can then be accepted by v.

($\Leftarrow$) Assume by contradiction that the conditions are not fulfilled and that v accepts the list of w, *i.e.*, v includes the list of w by computing its new list with `ant`—refer to Lines $14 - 16$ of Procedure `compute()`. That means that the list of w is compatible—refer to Lines $6 - 8$—, which contradicts the assumption. Then the nodes of $\texttt{list}_w^c$ will be propagated in the lists of nodes of $\texttt{list}_v^c$ and reciprocally. But at least one node $u \in \texttt{list}_v^c$ will see that a node $u' \in \texttt{list}_w^c$ is too far from it and reciprocally. Either u or u' will reject the lists of its neighbors that contain the too far node (depending on the priority between u and u') and either the group of v or the group of w splits (when a neighbor is rejected by u, it disappears from $\texttt{list}_u$, and then from $\texttt{view}_u$; it is then no more in H_v). □

Proposition 14 *For any execution e, for any configuration c_i in e, $\Pi_T(c_i, c_{i+1}) \Rightarrow \Pi_C(c_i, c_{i+1})$.*

Proof. Suppose that there exists a configuration c_i and a node v such that $\texttt{view}_v^{c_i} \not\subseteq \texttt{view}_v^{c_{i+1}}$. Then there exists a node u such that $u \in \texttt{view}_v^{c_i}$ and $u \notin \texttt{view}_v^{c_{i+1}}$. This cannot happen after u or v has added a new node in its view, thanks to the quarantine mechanism. This can only happen because either u or v removed a node from their views.

Without loss of generality, suppose that v removed a node x: $x \in \texttt{view}_v^{c_i}$ and $x \notin \texttt{view}_v^{c_{i+1}}$. If $x \notin \texttt{view}_v^{c_{i+1}}$, then (i) the quarantine of x is not null or (ii) x is not in $\texttt{list}_v^{c_{i+1}}$ or (iii) x is marked in $\texttt{list}_v^{c_{i+1}}$ (Line 31 in Procedure `compute()`).

(i) The first case is exclude because x was already in $\texttt{view}_v^{c_i}$.

(ii) In the second case, if v has not received the message of x while it received it before, then x left the neighborhood of v. Then, in configuration c_{i+1}, there is not path from x to v with only nodes of $\Omega_v^{c_i}$ and $d_{\Omega_v^{c_i}}^{c_{i+1}}(x, v) = +\infty$. Thus $\neg\Pi_T(c_i, c_{i+1})$ (*a neighbor left*).

(iii) In the third case, if x is simple marked, its list is not good while it was in configuration c_i, which is exclude (Line 3). If x is double-marked, this cannot happen after the compatibleList test (Line 7) because x was in $\texttt{view}_v^{c_i}$. If this happened after Line 19, then x sent a list with a too far node y having priority on v. If $y \notin \Omega_v^{c_i}$, then $y \notin \texttt{view}_v^{c_i}$. Then the quarantine of y is not null and no node of $\Omega_v^{c_i}$ has admitted y in its view. Therefore, thanks to Prop. 13, y would have never been propagated inside $\Omega_v^{c_i}$ until v, because of the compatibleList test (Line 6). Finally, if $y \in \Omega_v^{c_i}$, then the distance from y to v in configuration c_{i+1} is larger than `Dmax`: $d_{\Omega_v^{c_i}}^{c_{i+1}}(x, v) > \texttt{Dmax}$ and $\neg\Pi_T(c_i, c_{i+1})$. □

6 Conclusion

This paper introduces the best effort requirement to complete the self-stabilization for designing algorithm in dynamic networks. To illustrate this approach, a new problem inspired from VANET has been specified: the *Dynamic Group Service*. A best effort distributed protocol called `GRP` has been designed and proved for solving this problem in message passing. The algorithm is self-stabilizing and fulfills a continuity property whenever the dynamic allows it. The protocol has been implemented and its performances studied by simulation (see reference in Footnote 1 page 3). We believe that the best effort requirement is promising for building useful services in dynamic networks.

References

[1] A.D. Amis, R. Prakash, and D.H.T. Vuaong. Max-min *d*-cluster formation in wireless ad hoc networks. In *IEEE INFOCOM*, pages 32–41, 2000.

[2] Kenneth P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[3] J. Blum, A. Eskandarian, and L. Hoffman. Challenges of intervehicle ad hoc networks. *IEEE Transaction on Intelligent Transportation Systems,*, 5:347–351, 2004.

[4] O. Brukman, S. Dolev, Y. Haviv, and R. Yagel. Self-stabilization as a foundation for autonomic computing. In *The Second International Conference on Availability, Reliability and Security (ARES)*, pages 991–998, Vienna, April 2007.

[5] G.V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 4(33):1–43, 2001.

[6] A. K. Datta, L. L. Larmore, and P. Vemula. A self-stabilizing O(k)-time k-clustering algorithm. *Computer Journal*, 2009.

[7] S. Delaët, B. Ducourthial, and S. Tixeuil. Self-stabilization with r-operators revisited. In *Journal of Aerospace Computing, Information, and Communication*, 2006.

[8] Murat Demirbas, Anish Arora, Vineet Mittal, and Vinodkrishnan Kulathumani. A fault-local self-stabilizing clustering service for wireless ad hoc networks. *IEEE Trans. Parallel Distrib. Syst.*, 17(9):912–922, 2006.

[9] S. Dolev. *Self-Stabilization.* The MIT Press, 2000.

[10] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC)*, page 255, New York, NY, USA, 1995. ACM.

[11] S. Dolev, E. Schiller, and J.L Welch. Random walk for self-stabilizing group communication in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(7):893–905, 2006.

[12] B. Ducourthial. r-semi-groups: A generic approach for designing stabilizing silent tasks. In *9th Stabilization, Safety, and Security of Distributed Systems (SSS'2007)*, pages 281–295, Paris, novembre 2007.

[13] B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. *Theor. Comput. Sci.*, 293(1):219–236, 2003.

[14] R. Guerraoui and A. Schiper. Software-based replication for fault-tolerance. *IEEE Transaction on Computers*, 30(4):68–74, 1997.

[15] Arshad Jhumka and Sandeep S. Kulkarni. On the design of mobility-tolerant tdma-based media access control (mac) protocol for mobile sensor networks. In Tomasz Janowski and Hrushikesha Mohanty, editors, *ICDCIT*, volume 4882 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2007.

[16] Colette Johnen and Le Huy Nguyen. Robust self-stabilizing weight-based clustering algorithm. *Theor. Comput. Sci.*, 410(6-7):581–594, 2009.

[17] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[18] S. Kutten and D. Peleg. Fast distributed construction of small-dominating sets and applications. *Journal of Algorithms*, 28(1):40–66, 1998.

[19] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] L. D. Penso and V. C Barbosa. A distributed algorithm to find k -dominating sets. *Discrete Applied Mathematics*, 141(1-3):243–253, 2004.

[21] F.B. Schneider. Impliementing fault tolerant services using the state machine approach: a tutorial. *Computing Surveys*, 22(4):299–319, 2990.

[22] I. Stojmenovic. *Handbook of Wireless Networks and Mobile Computings.* John Wiley & Sons, 2002.

A Omitted proofs

A.1 Proof of Proposition 1 (Dmax)

Proof. Starting from configuration c_1, the system will reach in finite time a configuration in which every node has computed its list after expiration of its timer. After such a computation, the size of the lists is bounded by $\texttt{Dmax} + 1$ (because it is truncated at the $\texttt{Dmax} + 1$ position, Line 28). □

A.2 Proof of Proposition 2 (Exist)

Proof. Let $c \in e_{\texttt{Dmax}}$ be a configuration (Proposition 1). Let u be a node label such that $u \notin V$ and denote by U_k^c the set of nodes having u in their list at position k in configuration c. Consider the function $\phi(c)$ defined by $\phi(c) = \min\{k \in \mathbb{N}, U_k^c \neq \emptyset\}$ and $\phi(c) = \infty$ if $\forall k \in \mathbb{N}, U_k^c = \emptyset$. We prove that ϕ is continuously growing along the execution to be eventually equal to infinity forever.

Consider a node v in $U_{\phi(c)}^c$: v contains u at position $\phi(c)$ in its computed list and no node in configuration c contains u at a smaller position in its computed list. Until the next expiration of its timer, v cannot receive a list containing u in a smaller position than $\phi(c)$. Hence, the system will reach in finite time a configuration in which the node v has computed a new list that does not contain u at a position smaller than $\phi(c) + 1$. After a timer (fair channel Hypothesis), the system reaches in finite time a configuration in which the neighbors of v have received this list.

After finite time, any node $v \in U_{\phi(c)}^c$ will do the same. The system then reaches in finite time after configuration c a configuration c' in which $U_{\phi(c)}^{c'}$ is empty, meaning that $\phi(c) < \phi(c')$.

By iteration, ϕ is growing along the execution. Since the size of the lists is bounded by $\texttt{Dmax} + 1$ (Proposition 1), there exists a configuration c'' reached in finite time after c in which $\phi(c'') = \infty$, meaning that u does not appear anymore in the computed lists of the nodes forever. □

A.3 Proof of Proposition 5 (Double-marked edge)

Proof. Let v and w two nodes of G such that $d(v, w) = \texttt{Dmax} + 1$. Without loss of generality, we suppose that $pr(w) < pr(v)$. Suppose that there exists a path from v to w that does not contain any double-marked edge. By Proposition 4, there exists a neighbor u of v such that u sends to v a list containing w. The size of this list is larger than $\texttt{Dmax}$. There is two cases. (i) $u \notin \texttt{view}_v$. In this case, $\texttt{list}_u$ is replaced by $(\underline{\underline{u}})$. (ii) $u \in \texttt{view}_v$. In this case, v computes a list using the one sent by u. Since $d(u, v) > \texttt{Dmax}$, the resulting list is too long. Since $pr(w) < pr(v)$, the computation will be done again without the list provided by u, which will be replaced by $(\underline{\underline{u}})$. In the two cases, u is double-marked by v. Hence, any path from u to v will eventually contains a double-marked edge. □

A.4 Proof of Proposition 6 (Subgraphs)

Proof. By Proposition 5, there exists a suffix s_1 such that any path from u to v contains a double-marked edge. By Proposition 3, there exists a suffix s_2 included in s_1 such that for any configuration c in this suffix, $u \notin \texttt{list}_v^c$ and $v \notin \texttt{list}_u^c$. Then $u \notin H_v$ and $v \notin H_u$.

Let consider a node w such that $w \in H_v$ and $w \in H_u$. Then there exists at least one path from u to v containing w. The length of such a path is larger than $\texttt{Dmax}$. Then, by Proposition 5, it admits a double-marked edge, either on the subpath from u to w or from the subpath from w to v.

Now, let consider all the paths from u to v containing w; they all contain a double-marked edge. Suppose that for one path P_1, this double-marked edge is between w and v and for a second path P_2, it is between u and w. Then, by considering edges of P_1 from u to w and edges of P_2 from w to v, we obtain a path from u to v without any double-marked edge, which is a contradiction. Then, all paths from u to v containing w admit a double-marked edge, and this edge is always between u and w or always between w and v. Thus, w cannot belong to both H_u and H_v, meaning that there is no node w such that $w \in H_u$ and $w \in H_v$.

Hence, any execution reaches a suffix such that, for any configuration c in this suffix, H_u^c and H_v^c are distinct. □

A.5 Proof of Proposition 7 (Agreement)

Proof. By Proposition 6, for any execution, there exists a suffix such that, for any nodes u and v in G, if $d(u,v) > \texttt{Dmax}$, then the subgraphs H_u and H_v are distinct. Consider now two nodes w and v such that w belongs to H_v

By Proposition 4, for any execution, there exists a suffix such that, for any configuration c in this suffix, the identities of H_v will be in $\texttt{list}_w^c$.

By Proposition 3, for any execution, there exists a suffix such that, for any configuration c in this suffix, the $\texttt{list}_w^c$ contains only vertices of H_v.

After the end of the quarantine period, all the nodes in $\texttt{list}_w$ belong to $\texttt{view}_w$. Then the system reaches a suffix in which all the nodes of H_v and only these nodes appear in $\texttt{view}_w$, for any vertex $w \in H_v$. Hence, $\texttt{view}_v^c = \texttt{view}_w^c = \Omega_v^c$. This gives Π_A. □

A.6 Proof of Proposition 8 (Safety)

Proof. By Proposition 6, for any execution and any nodes u and v in G satisfying $d(u,v) > \texttt{Dmax}$, the subgraphs H_u and H_v will eventually be distinct. Hence, for any execution, there exists a suffix $e_{\texttt{safe}}$ such that, for any configuration $c \in e_{\texttt{safe}}$, for any vertex v in G, $\text{Diam}(H_v^c) \leq \texttt{Dmax}$.

Then, by Proposition 7, we have $\max_{x,y \in \Omega_v^c} d_{\Omega_v^c}(x,y) \leq \texttt{Dmax}$. This gives Π_S. □

A.7 Proof of Proposition 9 (Nee)

Proof. Let (u,v) be an external edge in a configuration c_i and assume that it is an internal edge in configuration c_{i+1}. This means that $\Omega_u^{c_i} \neq \Omega_v^{c_i}$ and $\Omega_u^{c_{i+1}} = \Omega_v^{c_{i+1}}$. Hence $nee(c_i) > nee(c_{i+1}) \Rightarrow ndg(c_i) > ndg(c_{i+1})$. □

A.8 Proof of proposition 10 (Not incr.)

Proof. Let $c \in e_{\texttt{agree}}$ be a configuration (Proposition 7). Let (u,v) be an internal edge in configuration c. Then we have $\Omega_u^c = \Omega_v^c$ and u is in $\texttt{list}_v^c$. In order (u,v) becomes an external edge, one of its extremity (say v) would have double-marked the other (in Procedure `compute()`). But this cannot happen after the `goodList` test (line 3) because $c \in e_{\texttt{subgraphs}}$. This cannot happen after the `compatibleList` test (line 6) because u is in already in $\texttt{view}_v^c$. □

A.9 Proof of Proposition 11 (Decreasing)

Proof. Let $c \in e_{\texttt{notincr}}$ be a configuration (Proposition 10). Starting from such a configuration, the *nee* function cannot increase. Suppose that Π_M is not true in c. Then, by definition of Π_M, there exists two neighbors nodes x and y with different views that could merge their groups without breaking Π_S. By fair channel hypothesis, a timer later the system reaches a configuration c' in which x (resp. y) has received the list sent by y (resp. x).

Without loss of generality, suppose that Ω_x has the smallest priority among all the subgraphs that can merge, and Ω_y has the smallest priority among all the groups that can merge with Ω_x.

During the `compute()` Procedure on x and y, the `goodList` tests are true because $c' \in e_{\texttt{notincr}}$ and then $c' \in e_{\texttt{safe}}$. The `compatibleList` test is true on both x and y because they cannot have change their list since configuration c. Hence we obtain: $x \in \texttt{list}_y$ and $y \in \texttt{list}_x$.

Since Ω_y has the smallest priority among the neighbors of Ω_x, no member of Ω_x can receive a message from a group with a smallest priority. Therefore x will never receive and then will never send to y a list with a too far node with a smallest priority than y one's. Hence y will never double-mark x and x will remain in the list of y.

Similarly, since Ω_x has the smallest priority among the groups that can merge, no member of Ω_y can receive a message from a group with a smallest priority. Therefore y will never receive and then will never send to x a list with a too far node with a smallest priority than x one's. Hence x will never double-mark y and y will remain in the list of x.

After `Dmax` timer, the list of y (resp. x) has reached any $u \in \Omega_x$ (resp. Ω_y) thanks to the fair channel Hypothesis. Moreover the quarantine of these new members reaches 0 and they are now included in $\texttt{view}_u$. Thus, the edge (x, y) becomes an internal edge.

Hence, starting from configuration c with $\neg\Pi_M(c)$, the system reaches in finite time a configuration c'' with $nee(c) > nee(c'')$. □

A.10 Proof of Proposition 13 (Compatible lists)

Proof. Let $c \in e_{\texttt{safe}}$ be a configuration (Proposition 8). Let w be the first node of Ω_w^c for which the list of ancestor's sets is received by v. Then, the only external edges between Ω_v^c and Ω_w^c known by v are those joining w (external edges are not propagated). Hence, without loss of generality, assume that only these external edges exist between the groups.

($\Rightarrow$) Assume that the conditions are fulfilled. Let $u \in a_v^k$ and $u' \in a_w^l$ be two nodes in the lists of v and w respectively. There exists at most two families of shortest paths from u to u', depending on the external edge used to reach w. Let P_1 be a path that includes the edge (v, w). It starts from u and joins v by k edges in the group of v, joins w by the edge (u, v) and then reaches u' by l edges in the group of u. Let P_2 be a path from the second family. It starts from u and joins a node $v' \in a_v^i$ by $|k - i|$ internal edges in the group of v, then joins w by the edge (v', w) and then reaches u' by l internal edges in the group of u.

The length of P_1 is bounded by $k + 1 + q$. But since P_1 is a shortest path, it is shorter to reach u' from u by joining a node of a_v^0 (*i.e.*, v) than by joining a node of a_v^i (such as v'). Hence we have $k \leq i/2$ and the length of P_1 is bounded $i/2 + 1 + q$, which is smaller than `Dmax` by hypothesis. The length of P_2 is bounded by $p - i + 1 + q$, which is also smaller than `Dmax` by assumption.

Hence, for any node u and u' belonging to the group of v and w respectively, there exists a path from u to u' with less than `Dmax` edges. The list of w is then compatible with the list of v, and can then be accepted by v.

($\Leftarrow$) Assume by contradiction that the conditions are not fulfilled and that v accepts the list of w, *i.e.*, v includes the list of w by computing its new list with `ant`—refer to Lines $14-16$ of Procedure `compute()`. That means that the list of w is compatible—refer to Lines $6-8$—, which contradicts the assumption. Then the nodes of $\texttt{list}_w^c$ will be propagated in the lists of nodes of $\texttt{list}_v^c$ and reciprocally. But at least one node $u \in \texttt{list}_v^c$ will see that a node $u' \in \texttt{list}_w^c$ is too far from it and reciprocally. Either u or u' will reject the lists of its neighbors that contain the too far node (depending on the priority between u and u') and either the group of v or the group of w splits (when a neighbor is rejected by u, it disappears from $\texttt{list}_u$, and then from $\texttt{view}_u$; it is then no more in H_v). $\square$