Design and Experimentation of a Self-Stabilizing Data Collection Protocol for Vehicular Ad-Hoc Networks – Extended Version

Yoann Dieudonné, Bertrand Ducourthial
CNRS Heudiasyc UMR6599
Université de Technologie de Compiègne (UTC)
{yoann.dieudonne,bertrand.ducourthial}@utc.fr

Sidi-Mohamed Senouci
ISAT - Université de Bourgogne
Orange Labs (FTR&D)
Sidi-Mohammed.Senouci@u-bourgogne.fr

ABSTRACT

In this paper, we present a protocol to collect data within a vehicular ad hoc network (VANET). In spite of the intrinsic dynamicity of such network, our protocol simultaneously offers three relevant properties: (1) It allows any vehicle to collect data beyond its direct neighborhood (i.e., vehicles within direct communication range) using vehicle-to-vehicle communications only (i.e., without passing through any infrastructure or one of its components); (2) It tolerates possible network partitions; (3) It works on demand and stops when the data collection is achieved. To the best of our knowledge, this is the first collect protocol having these three characteristics.

All that is chiefly obtained thanks to a specific tool, namely Operator ant, borrowed from the self-stabilization area which confers to our algorithm the nice property to recover by itself from topology changes. In addition to a theoretical proof of correctness, our protocol has been implemented and tested through the Airplug Software Distribution: Road and lab experiments are presented and discussed.

1. INTRODUCTION

1.1 Context

Following the current trend in Automotive Engineering, motor vehicles are equipped with more and more sensors in order to improve the safety of the driver and passengers as well as ride comfort.

Taken individually, local information provided by sensors gives an imperfect knowledge to cars and to their passengers. However, by comparing information collected from several vehicles, the knowledge can be built up. In other terms, by exchanging information and estimations from vehicle sensors and analog computers, information becomes more accurate and relevant and thus, the confidence level is increased. For instance, it has been demonstrated [5, 24] that collaborative localization uncertainty in groups of agents or vehicles is less compared to the situation where individual agents estimate their position separately. Various techniques have been proposed to integrate relative observations, like maximum likelihood estimation [16], particle filters [20], Kalman filters [23, 21], and Monte-Carlo simulation. Although the designs of the previous schemes have led to practical implementations and have demonstrated their effectiveness in certain settings through extensive simulations or experiments, before doing so, vehicles need to collect data from the whole network or to a lesser extent in their neighborhood. Through a network operator (NO), we can envisage to directly send and receive collected data on a large scale: Recipients are then vehicles but also the infrastructure managers and users of cellular networks. Various applications are possible, such as traffic estimation, average speed, available parking spaces, etc. While a mere 3G connection can transmit data arising from a single car, it seems more appropriate to aggregate data before forwarding them. This *modus operandi* helps to analyse information from vehicles, to contextually filter and merge them with respect to different selection criteria. As a result, it becomes possible to send only useful informations so as to save bandwidth. In this context, the architecture has to be compounded of a data collection protocol specific to highly dynamic networks best known as VANETs (which stands for Vehicular Ad Hoc Networks) associated with a protocol for forwarding aggregated data to the core network of the NO. In this paper, we aim to focus on the process of collecting data in VANETs.

1.2 Related works

Most of the data collections in vehicular ad hoc networks are tackled by using a mechanism of dissemination which is a process whereby each vehicle periodically broadcasts information about itself. A large number of data dissemination protocols have been recently proposed within the framework of VANETs [28, 22, 19, 2, 25, 18]. For instance, we can refer to opportunistic disseminations, such as [28], as well as geographical disseminations [19]. In the first case, propagation is performed with the use of opportunistic diffusion of data: In particular, messages are stored in each intermediate node and forwarded to every encountered node until the destination is reached. The second one consists in sending the message to the closest vehicle toward the destination until it reaches it. Likewise, many other types of dissemination exist: Thereby, we can mention peer-to-peer [19] and cluster-based dissemination [2]. Notwithstanding, all the disseminations, and by extension every data collection relying on them, are generally not upon request. Conse-

quently, information is recurrently diffused even if it is not necessary, leading to bandwidth waste. Moreover, since vehicles do not know which data will be relevant, they will tend to broadcast more than expected.

To circumvent this problem, data collection would have to issue from a demand started by some initiator. In a fixed network, such a data collection can be achieved with no difficulty through a wave algorithm, the PIF algorithm being certainly the most emblematic example [26]. The PIF algorithm, like most of general wave algorithms [27], works in two steps –both of them being suggestive of a wave.

The first step corresponds to a *broadcast phase* started by a sink which is called an initiator. During this step, the sink sends a broadcast message to all its neighbors. In particular, in the context of a data collection, this message has to contain the types of data to be collected. The neighbor, that receives the broadcast message for the first time, considers the node that has sent it as its parent and forwards the broadcast message to all its neighbors with the exception of its parent. Behaving like this, a spanning tree is built.

The second step corresponds to a *feedback phase* started by the leaves of the spanning tree. More precisely, when the leaves receive broadcast messages from all theirs neighbors, they send their own data to their parents as feedback for the broadcast message. Obviously, data are related to the types of data which appear in the broadcast message. The other nodes, which are not leaves, will receive the feedback messages with the collected data from their children. These nodes will join their own data to those of their children through a mechanism of aggregation and send them to theirs parents until the sink has taken in all the aggregated data of the complete network.

Unfortunately, due to their intrinsic dynamicity, the PIF algorithm is doomed to failure in dynamic networks such as VANETs. The deep reason stems from the fact that the PIF algorithm requires the spanning tree to remain invariant: However such a property cannot be fulfilled because links between nodes are subject to incessant breakages.

In [6], the authors bypass the problem by adapting a decentralized wave algorithm from [15] to a vehicular network. Nevertheless, their protocol relies on the assumption that the network remains permanently connected, otherwise their algorithm would be unable to terminate. In particular, it is assumed that no node can disappear but, in reality, this frequently occurs in dynamic networks such as VANETs.

1.3 Contribution

Our contribution is threefold:

- *Design of a self-stabilization data collection protocol.* We describe a protocol for collecting data in VANETs using vehicle-to-vehicle communications only. In our designing, every data collection follows a demand including, among others, types of data as well as the maximal duration and depth for the collect process. Contrary to [6], it is not required for vehicular networks to remain continuously connected. To achieve this, each vehicle recurrently confronts its local network view with the other views so as to update it by involving an *r-operator*. An r-operator is a specific tool which brings the nice property of self-stabilization to our algorithm. A self-stabilizing algorithm has the ability to recover by itself from an inconsistent state caused by transient failures. Since topology changes can be viewed as transient failures, our protocol is guaranteed to sustain the dynamicity of the vehicular network. In particular, every local view will tend to be quite accurate in spite of the dynamicity. In the rest of this paper, we will be led to clarify these notions and concepts but for more details about self-stabilization and r-operators, the reader may refer to [14, 3, 13].

- *Implementation of our protocol and road experimentations.* We prove that our algorithm named COL can be practically implemented. In this way, we built a prototype using the Airplug Software Distribution (ASD) in order to test it via real experimentions on road. ASD is a software suite which allows to develop distributed protocols suited to dynamic networks and allows to validate them via field or lab experiments [12, 17].

- *Lab experimentations over range of scenarios.* We show that COL can provide significant performances over a range of wireless communication scenarios. In different experiments, we varied several parameters notably the network dynamicity and the communication reliability to demonstrate the robustness of our algorithm.

1.4 Roadmap

The rest of this paper is organized as follows. First, some preliminaries are given in Section 2. Then, Section 3 introduces our data collection protocol and its proof of correctness. The proof of concept is presented in Section 5. Finally, we analyse performance evaluations in Section 6 before concluding the paper in Section 7.

2. PRELIMINARIES

In this section, we first introduce the specifications of the data collection problem considered in this paper. Next, we present two concepts, namely local network view and Operator ant, which are used in our protocol in Section 3.

2.1 Collecting data: specifications

There are many sources of data in a vehicular network that may produce interesting content, providing they have been gathered, aggregated and analyzed: the infrastructure itself when it is equipped, the vehicles that embed sensors, the humans (drivers, passengers, pedestrians...) using specific devices. To clarify the ideas, we mainly focus on collecting data produced by vehicles, while our algorithm may apply to other sources of data. The collect will then involve

inter-vehicle communication. Obviously several collect may run simultaneously but for sake of simplicity we consider a single collect, launched by a single vehicle called *initiator*.

A data collection application can be divided into four phases:

1. A preparing phase;
2. A gathering phase;
3. An aggregating phase;
4. A sending phase.

The **first phase** consists in elaborating the data to be collected. Indeed, while data can be almost instantaneously produced by sensors, it may be useful to consolidate it locally before the collect. Depending on the applications (and the kind of data), we may compare information from several sensors, compute a mobile average of the last values, compute the average of the values produced by close cars, and so on.

The data preparing phase can be either proactive or reactive, meaning that data can be prepared before the beginning of the collect or at the arrival of the first collect message. Hence, this phase involves either any vehicle that could be involved by a collect (proactive) or only those which are involved by the considered collect (reactive).

The collect algorithm can collect data of any *type*. The type of data to be used is given by the initiator and is included in the collect messages: parameter `typedt`. For instance, in our experiments, `typedt` is equal to `ident` to collect the id of vehicles or to `speed` to collect their speed (the first case allows to compute the density of the vehicles in a given area, while the second allows to compute the average speed in a given road).

It is worth noting that, depending on their type, data can be unstable (*e.g.*, `typedt=speed`), and can change during the gathering phase. The collect algorithm should then include a *conflict operator* denoted $\odot$ to deal with the case where a vehicle receives two different data (or more) coming from a single vehicle. This operator depends on the application; it may for instance return the lastest of the values in conflict or the smallest one, *etc.*

The **second phase** consists in gathering the data spread out in the vehicles to the initiator-vehicle. It is started by the *initiator vehicle* and involves necessarily a limited number of vehicles around the initiator. We limit the number of vehicles involved by using a `maxdst` parameter, representing the *maximal distance* in number of hops from the initiator. Indeed, each hop (vehicle-to-vehicle communication) increases the total duration of the collect as well as the number of messages in the network; `maxdst` is then an interesting parameter, impacting directly the performances. Note that we may use complex data type to limit the collect to a specific geographical area for instance (*e.g.*, ident of vehicles in the road r).

Note that, in a dynamic network, defining the termination of the algorithm using the distance from the initiator is not sufficient. Indeed, values may change and new vehicles may appear in the area and contribute to the collect with new values. Besides the difficulty, it is not always desirable. Indeed, for quickly meeting requests from vehicles about possible proximity events such as traffic jams, road covered in snow or covered in ice, fog, *etc.*, the collect would also have to be restricted in terms of duration. We then introduce two parameters: `maxdur` and `maxstb`. The first one gives the *maximal duration* of the algorithm on each node. The second one allows to optimize the duration; it gives the *maximal* number of successive *stable* values produced by a node before locally ending: if a node always produces the same value, it could locally end the algorithm before `maxdur`. By the way, even if new vehicles enter into the area of collect defined by `maxdst` during the collect, they will not delay the collect because other nodes will stop to propagate these values after `maxdur` units of time.

Parameters `maxdst`, `maxdur` and `maxstb` are used all together. Obviously, some combinations are not pertinent. For instance, too short values of `maxdur` may avoid to explore the network up to `maxdst` hops, and `maxstb` should be smaller than `maxdur` to be useful. Durations are measured in multiple of `aTimer` which is a constant for the duration of a timer.

The **third phase** begins when the collect is ended. The initiator can then aggregate the collected data by performing some computation (*e.g.*, average), depending on the application and the type of data. Note that in some cases, it could be possible to aggregate data during the second phase (distributed collect). Indeed, if the application consists in computing the smallest speed of any vehicle in an area, a vehicle could only send the minimum of all the received speed instead of sending all of them. While this may save space in messages, we prefer to separate the gathering phase from the aggregating phase for robustness. Indeed, suppose that a data is corrupted during the collect of the minimal speed and becomes null. Then it could become the final result. To the contrary, our algorithm is able to recover from such transient faults.

Finally, the **fourth phase** consists in sending the result to those who may request it. It is optional because the initiator vehicle may launch a collect for its own account. It could also share the result to close vehicles or send it to a server on the infrastructure. This last case requires a gateway to Internet. Collects for the infrastructure are either *push-based* or *pull-based*. Push-based collects are regularly started by some predefined initiator vehicles, such as road service vehicles that could be equipped with 3G device. Pull-based collects are initiated by the infrastructure to obtain information on a given area. For instance, the request is sent to a vehicle through a *Road-Side Unit* (RSU); the receiving vehicle becomes the initiator. It is not necessarily equipped with a 3G devices and it will have to discover a gateway to Internet for sending its result (3G device, RSU...).

In the rest of this paper, we focus on the distributed al-

gorithm (second phase) able to collect data in the vehicular network. For more details about the fourth phase, the inquiring reader is referred to [11].

2.2 Local view

Let start by some definitions in the aim of introducing the local view, which suppose to model the vehicular network as a dynamic directed graph, that is, a sequence of oriented graphs evolving in the time.

We define the *vicinity* of a vehicle v at date t as the set of vehicles which can directly send a message to v using the on-board vehicle-to-vehicle wireless device between $t -$ `aTimer` and t, where `aTimer` denotes the timer duration. Note that v may not be in the vicinity of u while u is in the vicinity of v.

At a given time t, a *vehicular network* can be viewed as a directed graph in which each vehicle is viewed as a vertex and such that there exists a directed edge from u to v if, and only if, u is in the vicinity of v at time instant t. In the remainder, $\mathscr{G}(t)$ will indicate the graph at Time t (or $\mathscr{G}$ when no ambiguity arises). We indifferently speak about vertex, node or vehicle for designing the vertices of a graph modeling a vehicular network.

As the vehicular network is dynamic, it is modeled by a sequence of graphs $\mathscr{G}_1, \mathscr{G}_2, \ldots$ where two consecutive graphs are not necessary different.

A *path* in a graph $\mathscr{G}$ is a sequence of consecutive directed edges of $\mathscr{G}$ $(u_0,u_1),(u_1,u_2),\ldots,(u_{n-1},u_n)$. The length of this path is n (number of edges). A *dynamic path* in a sequence of graphs $\mathscr{G}_1, \mathscr{G}_2, \ldots$ is a sequence of edges (u_0,u_1), $(u_1,u_2),\ldots,(u_{n-1},u_n)$ such that (u_0,u_1) belongs to $\mathscr{G}_1$, (u_1,u_2) belongs to $\mathscr{G}_2$, …, (u_{n-1},u_n) belongs to $\mathscr{G}_n$. The length of this dynamic path is n.

The *distance* from u to v in a graph $\mathscr{G}$, denoted by $dist(u,v)$, is the length of the shortest path from u to v. The *dynamic distance* from u to v at step k in a sequence of graphs $\mathscr{G}_1$, $\mathscr{G}_2,\ldots$, denoted by $ddist(u,v)$, is the length of the shortest dynamic path from u to v among those having the last edge in $\mathscr{G}_k$.

Let us denote by x_v the piece of data in vehicle v which has to be collected. Let consider a graph $\mathscr{G}$. We call *local view of depth p* of vehicle v the list $(N_0,N_1,\ldots,N_p)$ such that, for all $j \in \{1,\ldots,p\}$, N_j is the set of couples (u,x_u) satisfying $dist(u,v) = j$ and $N_0 = \{(v,x_v)\}$. Similarly, in a sequence of graphs $\mathscr{G}_1, \mathscr{G}_2, \ldots$, we call *dynamic local view of depth p at step k* of vehicle v the list $(N_0,N_1,\ldots,N_p)$ such that, for all $j \in \{1,\ldots,p\}$, N_j is the set of couples (u,x_u) satisfying $ddist_k(u,v) = j$ and $N_0 = \{(v,x_v)\}$. Figure 1 illustrates the definition of local view.

2.3 Operator Ant

In our protocol, each vehicle periodically updates its local view with respect to the local view of its neighbors, by using Operator ant [14, 13]. This operator belongs to the family of r-operator [10, 13]. When used for distributed computation on networks, these operators have interesting properties for fault tolerance, providing they fulfill some requirements. By modeling a local algorithm with such an operator, global properties of the distributed algorithm operating on the whole distributed system can be stated by simply checking the algebraic properties of the operator.

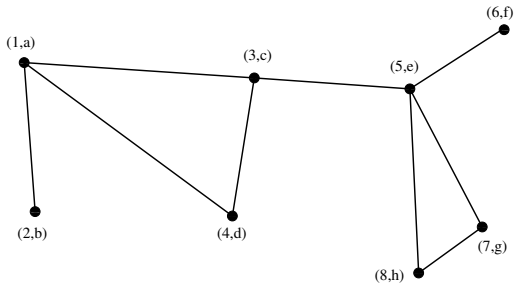

Figure 1: Examples of local views. For a depth of 2, the local view of Vehicule 1 is $(\{(1,a)\},\{(2,b),(3,c),(4,d)\},\{(5,e)\})$; the local view of Vehicule 3 is $(\{(3,c)\},\ \{(1,a),(4,d),(5,e)\},\ \{(2,b),(6,f),(8,h),(7,g)\})$.

Operator Ant has already be defined in previous work [14, 10, 13]. We give here the intuition behind its construction in order to explain its use for computing local views.

Let $\mathbb{S}$ be the set of well formed local views (views with empty sets N_i or with repetitive couples (x,x_v) are discarded by the nodes at the reception). We consider the operator $\oplus$ on $\mathbb{S}$ that merges two views while deleting needless or repetitive information so that a vehicle appears only once in a local view. To do so, for each vehicle v, we keep only the couple (v,x_v) having the lowest level *i.e.*, the leftmost. In case of conflict between two couples (v,x_v) and (v,x'_v) of same level, the ambiguity is resolved by using a conflict operator $\odot$, as explained in Section 2.1: only the couple $(v,x_v \odot x'_v)$ is kept. Providing that the binary operator $\odot$ is associative $(a \odot (b \odot c) = (a \odot b) \odot c)$, commutative $(a \odot b = b \odot a)$ and idempotent $(a \odot a = a)$, we can show that operator $\oplus$ is associative, commutative and idempotent on $\mathbb{S}$ (note that this is for instance the case when $\odot$ gives the latest or the smallest value produced by a vehicle in case of unstable data).

Since it is associative, commutative and idempotent, operator $\oplus$ defines an order relation $\preceq_\oplus$ on $\mathbb{S}$ by: $\mathscr{V}_1 \preceq_\oplus \mathscr{V}_2 \equiv \mathscr{V}_1 \oplus \mathscr{V}_2 = \mathscr{V}_1$. By the way, when using such operator, vehicles computes the smallest view of all those they received, prefering then small paths from ancestors instead of longer.

Nevertheless, each time a view is sent to a neighbor, its sets of couples have to be shift to the right because distances increase by one. This is done by an endomorphism r of $\mathbb{S}$, that insert an empty set at the beginning of the view: $r(\mathscr{V}) = (\emptyset,N_0,N_1,\ldots,N_p)$ where $\mathscr{V} = (N_0,N_1,\ldots,N_p)$.

Hence, any vehicle v periodically updates its local view $\mathscr{V}_v$ by computing the smallest view among $\mathscr{V}_v$ and $r(\mathscr{V}_u)$ for any view $\mathscr{V}_u$ sent by a neighbor u since the last local computation. The result operator is named *ant*; it is defined by:

$ant(\mathcal{V}_v, \mathcal{V}_u) = \mathcal{V}_v \oplus r(\mathcal{V}_u)$. We can show that it is a strictly idempotent r-operator inducing a partial order relation on $\mathbb{S}$ and the resulting distributed algorithm support transient faults [14, 10].

Let us take an example, that refers to Figure 1. Suppose that the view of Vehicule 1 is $\mathcal{V}_1 = (\{(1,a)\},\{(2,b),(3,c),(4,d)\})$ and suppose that it receives the view $\mathcal{V}_3 = (\{(3,c)\},\{(1,a),(4,d),(5,e)\},\{(2,b),(6,f),(8,h),(7,g)\})$ sent by Vehicule 3. Then Vehicle 1 computes its new local view by $ant(\mathcal{V}_1, \mathcal{V}_3) = \mathcal{V}_1 \oplus r(\mathcal{V}_3)$. This gives: $(\{(1,a)\},\{(2,b),(3,c \oplus c),(4,d)\}, \{\overline{(1,a)},\overline{(4,d)},(5,e)\},\{\overline{(2,b)},(6,f),(8,h),(7,g)\}) = (\{(1,a)\}, \{\overline{(2,b)},\overline{(3,c)},(4,d)\},\{\overline{(5,e)}\},\{(6,f),(8,h),(7,g)\})$.

Finally, to keep views of at most depth p, it is sufficient to truncate them just after the *ant* computation. Following the previous example, if $p = 2$, the list of Vehicle 1 becomes then $(\{(1,a)\}, \{(2,b),(3,c),(4,d)\},\{(5,e)\}$, which is the result given in Figure 1.

3. COL ALGORITHM DESIGN

In this section, we present our collect algorithm called *COL*, corresponding to the second phase in Section 2.1.

3.1 Algorithm Intuition

The intuition underlying the algorithm is simple and we briefly describe it here (see Algorithm 1 for details). At a high level, as soon as it is implicated in the collect, every vehicle periodically broadcasts its local view to all its neighbors. Of course, at the beginning of the data collection, only the initiator is concerned by the collect. However, during the process of the propagation of the messages, the number of concerned vehicles will grow whilst complying the maximum distance criterion (`maxdst`).

In the same way, every vehicle periodically recomputes its own view by applying Operator ant to the received local views. This recurrent process allows to take into account possible new vehicles implicated in the collect as well as possible topology changes due to the dynamic of the network. In particular, obsolete local views will be rectified thanks to Operator ant and its intrinsic property of self-stabilization.

When the data collection draws to a close, we consider that the result of the collect corresponds to the local view of the initiator.

So, to achieve that, our protocol considers two aspects namely handling the dynamic vicinity as well as collecting data as such.

3.2 Handling the dynamic vicinity

To handle the instability of the vicinity, each time a node v receives a message from a node u, it locally grants a lifetime of `maxloss` timers to u (lines 17 and 26). In this way, if ever v does not receive another message from u at the end of `maxloss` timers, v will consider u is no longer in its vicinity. More precisely, after a timer expires, the lifetime of u will be decremented by one (line 31). When the lifetime reaches the value of zero, all the data relative to u is erased from v (line 33).

Algorithm 1: **Collect protocol COL, for any node** v

```
1   Starting_action(typedt, maxdst, maxdur, maxstb):
      ▷ We supposed a single initiator for sake of simplicity.
      Other nodes cannot start the distributed algorithm.
2     if v is not initator or col_active == true then return
3     col_active ← true            ▷ A collect is now running
4     col_id += 1                  ▷ Id of the collect
5     col_initiator ← v
6     col_param ← (typedt, maxdst, maxdur, maxstb)
7     local_data ← item of data of v of datatype typedt
8     local_view ← {(v, local_data)}
9     count_dur ←0                 ▷ Count until maxdur
10    count_stb ← 0                ▷ Count until maxstb
11    tab_views ← ∅                ▷ List of last received views
12    send( col_id, col_initiator, col_param, local_view )
13    start timer with duration aTimer

14  Upon message arrival:
15    receive( rcv_id, rcv_init, rcv_par, rcv_view ) from u
16    if col_active == true and col_id == rcv_id then
        ▷ Message for the current collect
17      tab_lifetime[u] ← maxloss
18      tab_views[u] ← rcv_view    ▷ Store the received view
19    else if rcv_id > col_id
        ▷ New collect
20      col_active ← true          ▷ A collect is locally running
21      col_number ← rcv_id        ▷ Store the collect id
22      col_initiator ← rcv_init
23      (typedt, maxdst, maxdur, maxstb) ← rcv_par
24      count_dur ←0               ▷ Count until maxdur
25      count_stb ← 0              ▷ Count until maxstb
26      tab_lifetime[u] ← maxloss
27      tab_views[u] ← rcv_view    ▷ Store the received view
28      start timer with duration aTimer
29    end if
      ▷ Other messages are ignored.

30  Upon timer expiration:
      ▷ Detecting neighbors disappearance
31    tab_lifetime[u] -= 1 for any u in tab_lifetime
32    for each u such that lifetime[u] == 0 do
33      Delete entry u in tab_lifetime and tab_views
34    end for
      ▷ Computing the new local view
35    old_local_view ← local_view
36    local_data ← item of data of v of datatype typedt
37    local_view ← {(v, local_data)}
38    for each u such that tab_views[u] exists do
39      local_view ← ant(local_view, tab_views[u])
40    end for
41    Truncate local_view to the first maxdst elements
      ▷ Termination detection
42    count_dur += 1
43    if old_local_view and local_view are equivalent then
44      count_stb += 1
45    else
46      count_stb ← 0
47    end if
48    col_active ← false
49    if col_initiator ∈ local_view then
        ▷ Valid view regarding the collect
50      send( col_id, col_initiator, col_param, local_view )
```

```
51        if count_stb ≠ maxstb and
                              count_dur ≠ maxdur then
52            restart timer with duration aTimer
53            col_active ← true
54        else if col_initiator == v
              ▷ End of the collect on the initiator. Aggregating
              phase, see Section 2.1)
55            Compute the final result using local_view
56            col_active ← false        ▷ Allow to start a new collect
57        end if
58    end if
```

3.3 Collecting data

3.3.1 Starting the collect

As mentionned above, every data collection is triggered by a single node called the initiator. As soon as the initiator decides to start a collect, it sends a message in its neighborhood made up of four fields (line 12):

- collect number (col_id);
- identity of the initiator (col_initiator);
- collect parameters (col_param) ;
- its current local view (col_view).

Let us detail these fields. Concerning the identity of the initiator, it remains here unchanged as stated in Section 2.1 (for sake of simplicity, but several data collection could be carried out simultaneously). By contrast, the collect number never ceases to change: it is incremented by one with every new collect.

Concerning the collect parameters, these are selected by the initiator and are 4 in number (see Section 2.1): `typedt` (datatype to be collected), `maxdst` (maximal distance from the initiator for which the collect is desirable), `maxdur` (local maximal duration), `maxstb` (local maximal duration in case of stable view).

At the time of the first emission from the initiator, its vicinity knowledge is reduced to the empty set. In particular, its local view contains only its identity and its local data. However, it is to be expected that its local view will expand.

3.3.2 Receiving a message

At the arrival of a message (line 15), node v looks into the received collect number rcv_id so as to check whether the message is relevant or not. To do so, rcv_id is compared to col_id, the number of the last collect that v has taken part in.

In case v is currently involved in a collect (col_active is true on v), if rcv_id and col_id match, then the received message has to be taken into account for the current collect (line 16).

In case v is currently involved in a collect and receives a message belonging to a new collect (rcv_id $>$ col_id, see line 19), it resets its data regarding the collect. This may happen in case the initiator ends the collect before v (for instance, it loosed connectivity with its neighbors and obtain rapidly a stable result).

In case v is not currently involved in a collect (col_active is false), the message sent by u is taken into account by v only if it belongs to a new collect: rcv_id $>$ col_id (line 19). By the way, a node which has localy terminated the current collect will discard messages from this collect (and from older collects).

When the message is accepted, the sender view is stored (lines 18 and 27). When the receiving node enters into a new collect, the parameters are stored and the controlling variables are reset (lines 20-25), and the timer is started (line 28).

3.3.3 Periodic computation

Node v computes a new local view at timer expiration. The new view of v depends only on views sent by neighbor nodes considered still present in its vicinity (*i.e.*, nodes for which lifetime is not null, as explained in Section 3.2). To do so, Operator ant introduced in Section 2.3 is used (line 39). The resulting new local view is truncated to the first `maxdst` elements in order to respect the distance from the initiator (line 41).

The last step consists in detecting the termination of the collect (lines 42-58). The number of computations since the beginning of the current collect is increased (line 42); the number of successive computations with the same result is increased or reset, depending on the successive views (lines 43-47) . Then, col_active is reset (line 48) and will be set to true only if the collect has to continue (line 53).

If the initiator is not in the new local view, then Node v is not concerned by the collect: it is too far from the initiator. In that case, Node v no longer participates in the data collection. In the converse case, the message is broadcasted in the neighborhood.

If the number of computations (count_dur) has not reached `maxdur` and the number of identical successive views (count_stb) has not reached `count_stb` (line 51), the timer is restarted for a new computation (line 52). In the converse case, the node has locally terminated. If it is the Initiator, the final result is obtained. Phases 3 and 4 can begin, as described in Section 2.1: aggregating the collected data, sending the result.

4. CORRECTNESS

The COL algorithm is able to build local views in the network. In this section, we explain the interest of local views, both for the collect and the correctness. Next, we give the intuition of the correctness. Then we give the sketch of proof, by focusing first on the case where our algorithm runs forever (*i.e.*, no termination test), before studying the general case (with termination).

4.1 Correctness intuition

We explain with examples how our algorithm builds local

views of the network. Figures 2 to 4 illustrate the building of the local views at the beginning of the COL algorithm in a simple network composed of three nodes. Thanks to *ant* operator, local views are updated after each timer expiration to converge to the final result in Figure 4.

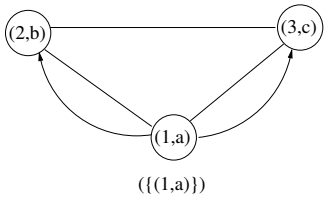

Figure 2: Vehicle 1 is the initiator. At the beginning of the collect, its local view is reduced to itself.

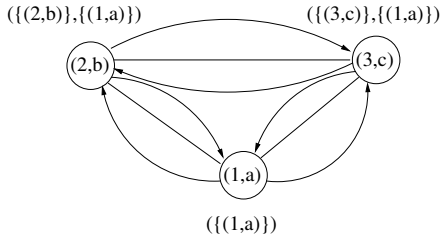

Figure 3: Vehicles 2 and 3 receive a message from 1. From now, all the vehicles are implicated in the data collection. Each of them builds its local view before rebroadcasting it.

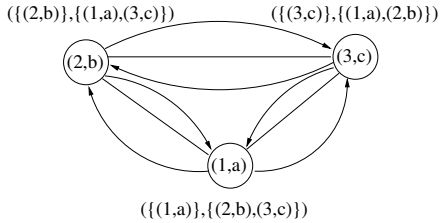

Figure 4: One timer expiration later, every vehicle updates its local view with respect to the other received local views.

Figures 5 to 7 illustrate the fact that the local views are corrected after a topology change: the connection between Nodes 2 and 3 in Figure 4 no longer exists in Figure 5. This change is detected when the lifetime of the missing neighbor (initialized at `maxloss` at each message arrival, and decremented at each timer expiration) reaches 0.

An example illustrating the fact that the local views are eventually corrected following the entry of a new vehicle, namely Vehicle 4, is depicted in Figure 8 to 12. Vehicle 4 will rapidly build its local view thanks to views received from neighbors. Reciprocally, its neighbors will incorporate its data, that will propagate from one hop per timer expiration.

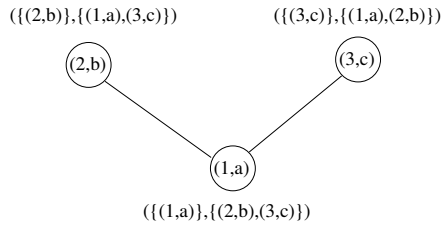

Figure 5: A topology change arises. Vehicle 2 and 3 are no longer neighbors. They can no longer directly communicate.

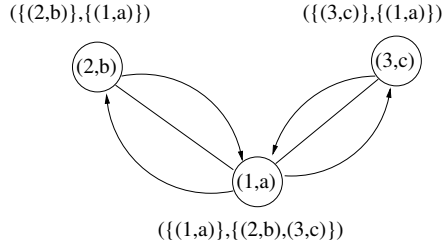

Figure 6: After `maxloss` timers, Vehicle 2 and 3 eliminate respectively 3 and 2 from their respective local view.

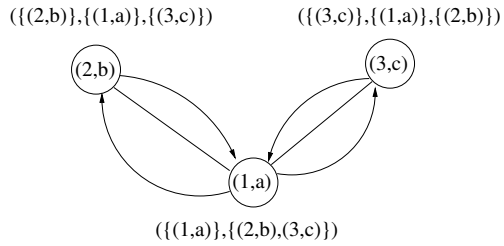

Figure 7: Two timer expirations later, all the local views become correct again.

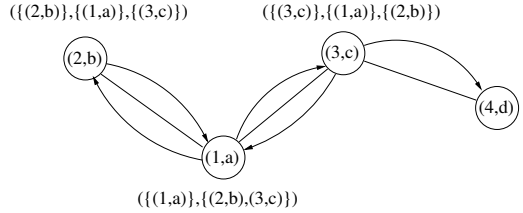

Figure 8: A new node, namely Vehicle 4, appears inside the network. It is not yet implicated in the collect. However, being a neighbor of Vehicle 3, it receives a message from it.

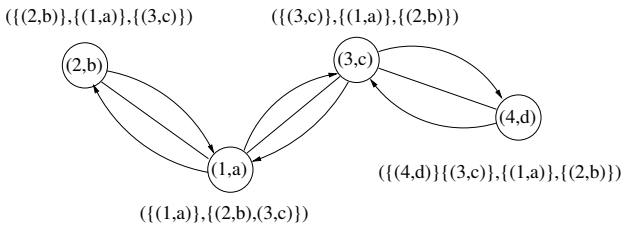

Figure 9: Vehicle 4 is now implicated in the collect. It computes its local view according to the one sent by Vehicle 3. Once completed, it broadcasts its local view to Vehicle 3.

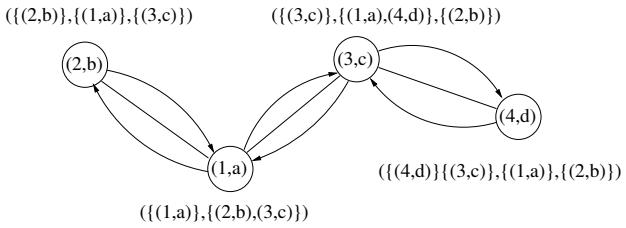

Figure 10: Vehicle 3 updates its local view and sends it to Vehicle 1.

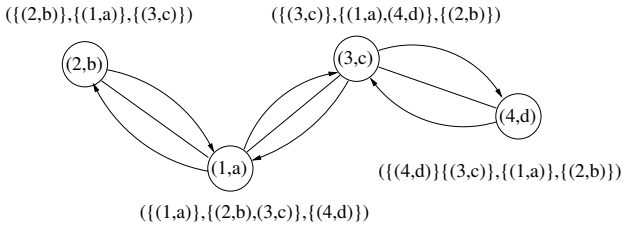

Figure 11: Vehicle 1 in its turn updates its local view and broadcasts it.

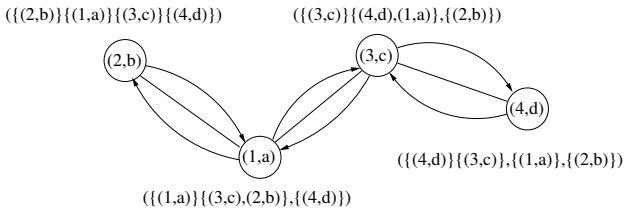

Figure 12: All the local views are now correct.

4.2 Self-stabilization property

Self-stabilization is a concept of fault tolerance in distributed system first introduced by Dijkstra in [8]. Several equivalent definitions exist in literature but we can remember the following one:

DEFINITION 4.1 (SELF-STABILIZATION). *[9] An algorithm $\mathscr{A}$ is said to be self-stabilizing if and only if:*

1. **(Convergence).** *Starting from any state, it is guaranteed that $\mathscr{A}$ will allow the system to eventually reach a correct state.*
2. **(Closure).** *Given that the system is in a correct state, it is guaranteed to stay in a correct state while executing $\mathscr{A}$, provided that no fault happens.*

In particular, a self-stabilizing algorithm has the ability to recover by itself from an inconsistent state caused by transient failures. It could then run without any initialization.

By *transient failure*, we intend unwanted disturbing of limited duration in the time, such as packet loss, volatile memories modification or messages alteration. It is worth noting that a topology change leads to an inconsistency in the neighborhood knowledge (and local views) of all impacted nodes. By the way, any topology change can be modeled as a transient failure and self-stabilization algorithms have the nice property to support the network dynamic.

However, in order to fulfil its self-stabilizing requirement, an algorithm should be able to run until the convergence happens (which should occur in finite time). As our algorithm termination is determined by the `maxstb` and `maxdur` parameters, the COL algorithm may not have enough time to recover from a transient failure if those parameters would not have well set. Then, we first consider the properties of the COL algorithm during an infinite execution:

PROPERTY 4.2. *The COL protocol is self-stabilizing and builds a local view of the network centered on the initiator, providing it runs forever.*

This property is due to the fact that the aforementioned collect protocol is mainly based on Operator ant which confers to COL the property of self-stabilization. Indeed, Operator ant leads to a large range of self-stabilizing tasks, such as computing local views, in a kind of distributed systems which admit bounded communication links. As stated in [7], since wireless communications can be viewed as bounded links and seeing that topology or data changes can be viewed as transient failures, the nice property of self-stabilization can be directly extended to the present framework.

Basically, the Operator *ant* relies both on the Operator $\oplus$ and the endomorphism r (see Section 2.3). Roughly speaking, the first one tends to decrease the values as the second one tends to increase the values, leading to a stable state satisfying the specification.

Indeed, as $\oplus$ defines an order relation, the local computations performed by each node can be assimilated as a kind

of minimum computation. By the way, any too large unlegitimate value is eventually withdrawn from the network. Moreover, since the r-operator is strictly idempotent, it satisfies $x \prec r(x)$. Hence, each time a value progresses in the network, it is incremented by the endomorphism. By the way, any too small unlegitimate value is eventually withdrawn from the network. Only the values which are regularly broadcasted are kept in the network, meaning that local nodes output will eventually be legitimate.

Note that, in the special case where there are some data changes (e.g., vehicle speeds), old values are replaced with new values provided that the conflict operator is wisely chosen. Some examples of the self-stabilizing features are depicted in Figures 2 to 12.

4.3 Dynamic network

The self-stabilizing property of the COL algorithm ensures that it can support any transient failure providing it has enough time to converge. However, in order to obtain a result in bounded time, the COL algorithm includes a termination detection using both `maxdur` and `maxstb`. Property 4.2 will be fulfilled only for wisely chosen values of such parameters.

Nevertheless, even when the convergence has not be reached, the algorithm outputs a *dynamic* local view centered on the initiator:

PROPERTY 4.3. *The COL protocol builds a dynamic local view of the network, centered on the initiator (providing there is no corruption of volatile memories).*

Indeed, if u is in the view of Initiator v, then u has been sent by a neighbor of v during the previous timer. If such ancestor had u in its view, then either it is u itself or one of its neighbors outputs u a timer before... In any case, a dynamic path exists between u and v in the sequence of graphs modeling the evolving topology.

Besides its interest for self-stabilizing properties, local views present other several advantages, from the point of view of data collection algorithm.

First, it includes the data to be collected. Second, it includes a topological representation of these data, rich information that can be exploited by the aggregating phase. Let us give some examples among others. (i) A confident coefficient could be affected to the result depending on the size of the local view, on the redundancy of some values inside the view, and so on. (ii) Divergent data could be filtered using a distance-based filter, giving more relevance to close data compared to the farthest in case of doubt. (iii) The initiator could consolidate the collected data by computing a weighted average, where weights depend on the distance to the initiator.

5. ROAD EXPERIMENTS

To establish a proof of concept of our protocol, some road experiments were conducted using the Airplug Software Distribution (ASD). ASD is a software suite which is dedicated to the design of distributed applications in highly dynamic ad hoc networks, such as vehicular networks. In particular, it allows field as well as lab experimentations. In this section, before focusing on the proof of concept, we describe the software component of ASD, namely Airplug-road, which is especially devoted to road experiments.

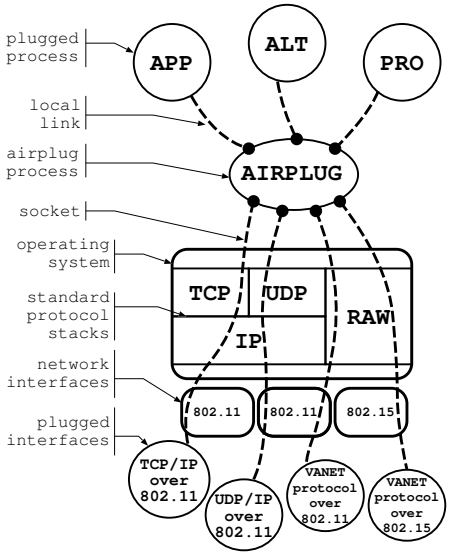

Figure 13: A node in the airplug architecture.

5.1 Architecture for road experiments

5.1.1 Process-based architecture

In order to obtain a light, portable and robust distributed framework, the Airplug architecture relies on the facilities given by standard operating systems: resources allocation, process scheduling, real-time management... This avoids any redundancy between the framework and the operating system, and allows to take benefit of any improvement in these fields (including real time management). It is expected a portable POSIX operating system with process management and memory protection for robustness issue.

The framework comprises a core program –called Airplug-road– one for each mobile node, that runs in a standard process on top of the operating system. By not including any part of the framework in the kernel nor in the applications, the independence with the operating system as well as with the applications programming is enforced.

The Airplug architecture accepts either local or distributed applications. A local application does not have any interaction with remote applications. A distributed application is composed of several instances of the same program, running in different mobile nodes, and exchanging messages. Local applications as well as local instances of the distributed applications run in separate processes with their own memory space. This enforces the applications independence as well as the reliability: An application may be bogus with very

few impact on the rest of the system. Moreover, this allows to subcontract the scheduling of the applications as well as the real-time management: The framework should just set the priorities regarding parametric rules (eg. context-aware heuristics) and the operating system does the rest.

All these processes are launched by airplug, which creates descendant processes, called plugged processes. By this way, airplug is easily informed of the problems of an application by catching the related signals sent by the OS to the parent process (eg. abnormal termination).

5.1.2 Inter-process communications

To fit with the asynchronous opportunistic network, the protocol is based on asynchronous messages passing. In order to minimize the requirements for the applications development, inter-process communications are done with messages through standard IO. Indeed, any process owns by default a standard input stdin, a standard output stdout (and a standard error output stderr). This functionality is then supported by any programming language, and it gives no requirement on the application programming. The standard IO are sufficient to perform the inter-process local communications, and it can easily be extended to ensure inter-process remote communications (between distant nodes). Moreover, such a communication scheme allows easy standalone use of the applications (without airplug) and it permits to reuse existing applications.

For each plugged process, standard input and output are redirected from and to airplug via bi-directional connected communication links called in the following local links (see Figure 1). There is one local link per plugged process. By this way, each time a plugged process writes on its standard output, airplug will receive the data via the related local link. And each time airplug writes on a local link, the related plugged process will receive the data. The airplug program scrutinizes the local links to receive the data from the plugged process and forward them to the destinations (local or distant process specified by the sender).

Hence airplug represents a kind of "bus" between all the plugged processes. The network interfaces are also connected to this bus so that any application on the top of an airplug instance in the vicinity are connected to a common applicative bus. This bus can be extended by means of multihops communications. It is important to note that this bus is very simple and can then be efficiently implemented by avoiding any kind of unwanted processes synchronization. This simplicity is well adapted to the opportunistic networks: it provides quick communication with very few common conventions and without any global management (eg. services directory); it then supports rapid extension or reduction depending on the nodes movements. Note that, in addition to the efficiency, this communication scheme preserves the language independence: The more adapted programming paradigm can be chosen to build an airplug compatible application i.e., object oriented or not, multi-threaded or not, interpreted or compiled and so on. In a nutshell, the language independence allows to take benefit of any language and compiler improvement, and to remain open to new future programming paradigm.

5.1.3 Networking integration

In the Airplug architecture, the network interfaces are accessed through airplug, and are called plugged interfaces: An example is depicted in Figure 13. The plugged interfaces are managed as the plugged processes, to the exception that they are connected to airplug via some sockets. Hence, the network is addressed by the applications in the same way they address a message to another application, simply by writing to their standard output. The airplug program receives the data sent by the plugged processes and sends them to the desired plugged interface through the related socket.

By using several plugged interfaces, several network devices can simultaneously be accessed, an several protocols can be used. For instance, sockets may be open for the Bluetooth interface and other to some WiFi interfaces. Some of them could use the TCP/IP stack while some other could perform broadcast over UDP. By allowing to use the network stacks included in the kernels, airplug ensures the compatibility with any existing network such as Internet, and can take benefit of any improvement of such protocols. For instance, IPv6 may be used by including it in the kernel, and by opening the corresponding socket.

But this architecture is also open to any new communication scheme. By using raw sockets, the link layer can directly be accessed through a plugged interface. This can advantageous be used to broadcast in the neighborhood by avoiding the IP and UDP headers. Moreover, this allows to implement in user space new protocols inside plugged processes. Such a process will receive the messages from the applications that want to send their data using the new protocol. It will then send messages through a raw socket to the remote hosts: An example of that is depicted in Figure 14. Hence, any by-pass of the Internet stack as well as cross-layering solution can be designed by simply using the inter-applications facilities of airplug.

5.2 Proof of Concept: Road Experiments

In addition to the theoritical proof described in the previous section, we aim to establish a proof of concept. In the context of VANET, a proof of concept essentially consists of implementations and road experiments in order to demonstrate its feasibility in practice. By definition, a proof of concept may be not complete and is usually restricted to a small number of scenarios. So, to demonstrate the feasibility of our algorithm, we have implemented our protocol in Tcl/Tk in order to test it on road through airplug. The resulting application is simply named COL.

For this purpose, five cars have been mobilized. Within each of them, exactly one PC (Dell mini-9 Model DP118) under Ubuntu (v8.04 Hardy Heron) and running Airplug as

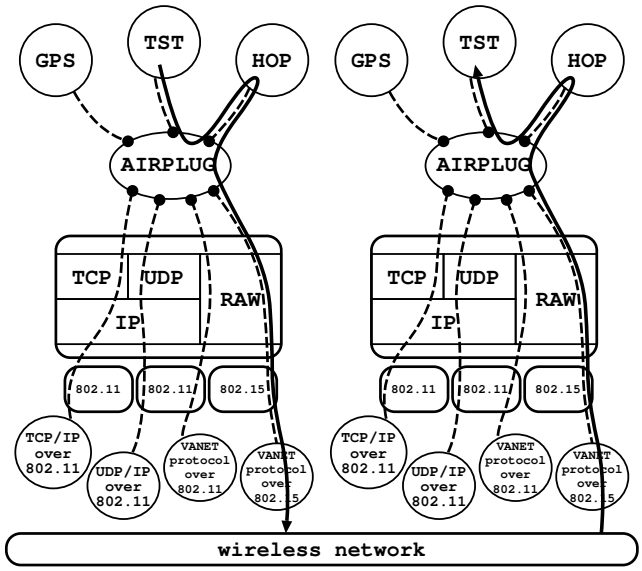

Figure 14: Integration of a new protocol in the airplug architecture. Here the HOP protocol is used by the ALT application.

Figure 15: Road experiments used experimental vehicles of the lab as well as standard cars only equipped with the mini Dell PCs.

Figure 16: Experimental platform composed of mini Dell under Linux, external WiFi cards, external antennas and GPS.

well as the COL application was installed (see Figure 16). All the PCs are equipped with an external WiFi card with USB connectors (Alfa AWUS036EH), allowing to connect an antenna on the roof of the vehicles (D-LINK ANT24-0700, 2.4 GHz, 7 dBi, omni-directional).

For this realization study, we have considered a scenario corresponding to a single convoy with five vehicles travelling at 90 km/h. The inter-vehicle distance is around 50 meters, corresponding to the expected security distance (equivalent to 2 seconds). Within the convoy, the car situated at the head of the convoy is the initiator: it successively launches several data collection according to the following parameters.

1. `aTimer` = 1000 ms
2. `maxloss` = 2
3. `typedt` = Ident (*i.e.*, collect of the vehicles' identity)
4. `maxdst` = 4
5. `maxdur` = 10

Parameters `aTimer` and `maxloss` are set in an arbitrary way but they are however large enough so that messages can be exchanged and so that the local views are not updated in an untimely manner. Concerning the datatype to collect, we just focus on the identities of the vehicles. The maximal number of hops (`maxdst`) is fixed to 4 in order to involve all the vehicles in the data collection notably the tail of the convoy. The maximal duration (`maxdur`) is fixed to 10 timer expirations: In this way, the duration of the collect is long enough to possibly allow time for possible data from the tail of the convoy to reach the initiator.

In the aforementioned scenario, the initiator has always been able to collect all the identities including its own, despites the dynamic of the vehicular network. Consequently, these experimentations constitute the proof of concept of our protocol, showing the feasibility of COL in practice, and complementing the theoritical proof described previously. These experiments lead to a movie, available online [1].

6. PERFORMANCE EVALUATIONS

So, what of the performance evaluations? As a matter of fact, our algorithm does not really lend itself to any performance evaluation for two main reasons. The first comes from the fact that several time parameters such as duration of the collect, maximum distance, *etc.* are left to the discretion of the initiator: However, these parameters directly affect the performance of our algorithm. The second stems from the fact that our protocol is qualitatively different from the existing protocols: In particular, it supports possible network partitions contrary to existing studies.

Nevertheless, to get the measure of our data collection, COL was tested in lab by replaying some real road conditions through Emulator Airplug-emu [4]. Airplug-emu is a component of ASD which allows to emulate vehicular ad hoc networks. In particular, it may reproduce road experiments without further developments of the studied protypes tested via Airplug-road. So, in the remainder, we analyse the impact of three different criteria on COL, communication reliability, timer duration and *MAXLOSS*.

6.1 Airplug-emu

Overview. Airplug-emu aims to perform realistic experiments of protocols and applications designed for vehicular

networks. With Airplug-emu, the applications and protocols to be studied run on independent processes as they do during road experiments, without any modification. Airplug-emu handles all the communications, either intra- or inter-vehicle, by using the shell facilities since applications run in independent processes and communications rely on standard input/output (refer to Subsection 5.1). This application is written in Tcl/Tk and runs on a Linux PC. Several computers can be used in order to introduce real links instead of emulated ones (hybrid emulation).

Scenario. The scenario of the test is described in an XML file, that indicates the number of vehicles, the size of the geographic area, the applications and protocols running in each vehicle, the trajectory of each vehicle, and so on. The trajectories are real positions obtained from our several field experimentations. However, Airplug-emu can accepts other input for node mobility, produced by traffic generators. The Network Simulator format is also accepted.

Link. EMU reads the position of each node in the network with a user-defined frequency. The communication links are determined by the wireless *communications range* (user-defined) and a random factor hazard which plays the role of *link reliability*. If the distance between two vehicles is less than range $\times$ hazard, then there is a link. The hazard (uniform random law) permits to add (or not) a variation in the antenna scope (to avoid perfect discs). It is also possible to use node-specific ranges, which is useful for some VANET security studies.

Network emulation. For each vehicle, EMU launches the applications and protocols specified in the XML file with the related command line, so that they run in independent processes. The standard input of these processes are connected to a reception process RCP and their standard output are connected to a directional process DIR. The first one receives all the inter- and intra-communications. The second one forwards the messages either for local applications or for neighbor vehicles through the gateway process GTW. The RCP process is a Tcl shell script that forwards intra-vehicle messages. The GTW process is implemented with the cat command. As previously explained, it is used to change the inter-vehicle connections without packet losses (if a perfect network is desired): first GTW is frozen, next the inter-vehicle links are changed, and then GTW is unfrozen and the messages waiting in its input are sent without losses. The DIR process is a Tcl shell script that analyzes the header of the messages sent by the local applications in order to determine whether they should be sent locally (keyword LCH) or to neighbor cars (keyword AIR) or both (keyword ALL).

Realistic emulation. All inter-process links including inter-vehicle links (from a GTW process on vehicle A to a RCP process on vehicle B) rely on shell named pipes. In order to reproduce the conditions of communication observed on the road, the RCP process can delay or lose inter-vehicle messages. It then accepts two parameters (delay and lossrate);

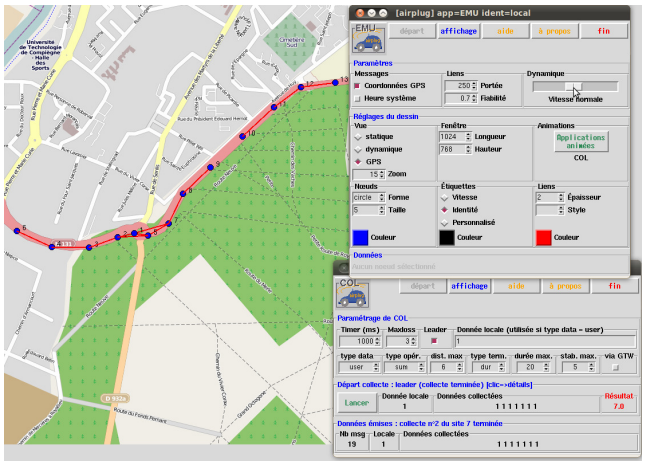

Figure 17: Emulation of the *COL* application on a convoy of 13 vehicles generated by the *GPS* application from a real *GPS* trajectory obtained on road N131, Compiègne, France. We can see the *COL* application running on Vehicle 7, as well as parameters of *Airplug – emu* and *COL*.

Such values can be measured during road experiments. The dynamics of the network can be varied thanks to a *dynamic* parameter in order to study the robustness of protocols in function of different levels of network dynamics. This parameter increases or decreases the vehicle locations updates whereas messages always take the same time to reach their destination.

6.2 Evaluation Results

For the tests in the laboratory, the mobility of the vehicles has been emulated through Airplug-emu, by using logs of real GPS positions, obtained on the road. More precisely, we consider a scenario in which 13 vehicles runs in Compiègne streets, a city in France. Each of them run COL and one of them plays the role of the initiator (refer to Figure 17). The communication range is fixed at 250 meters. The inter-vehicle distance varies between 200 and 350 meters. So, since inter-vehicle distance may be greater than communication range then the convoy is not always connected. Within the convoy, the car situated at the middle of the convoy is the initiator which successively launches data collections according to the following parameters:

1. `typedt` = ident (collecting the vehicles' identities)
2. `maxdst` = 6
3. `maxdur` = 20

The `maxdst` parameter is fixed at 6 in order to involve all the vehicles in the data collection notably the tail and the head of the convoy. The `maxdur` parameter is fixed at 20 timer expirations: In this way, the duration of the collect is long enough to possibly allow time for possible data from the tail as well as from the head of the convoy to reach the initiator when the convoy remains connected.

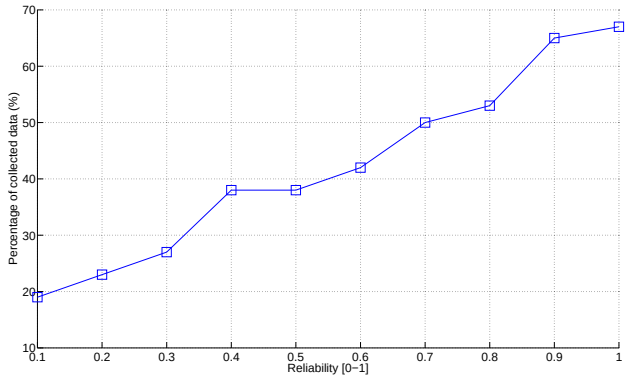

Figure 18: Percentage of collected data as a function of reliability of communication links, for a convoy scenario

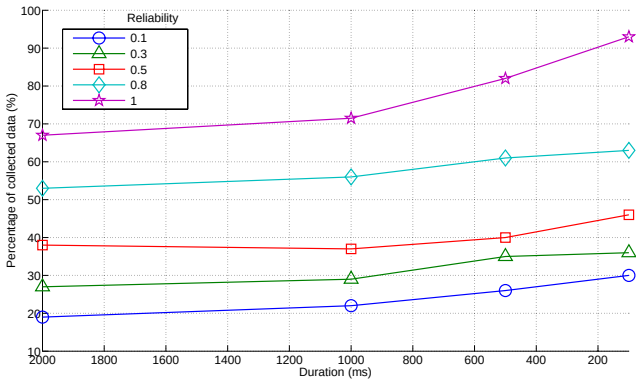

Figure 19: Percentage of collected data as a function of timer duration `aTimer`, for a convoy scenario

In this section we propose to study the impact of three parameters namely the link reliability (through the emulator), the timer duration (`aTimer`) and the initial lifetime of a neighbor (`maxloss`).

Link reliability.

To study the impact of the reliability of communication links on the number of collected data, for each level of reliability we run 50 simulations and we recorded the average percentage of collected data. The parameters `maxloss` and `aTimer` are respectively fixed at 3 and 2000 ms. Figure 18 plots the average percentage of collected data as a function of the reliability of communication links. This figure shows that the more reliable the links are, the greater the percentage of collected data is. So, the figure illustrates that our data collection protocol still works even when there are lost messages. Note that, due to recurrent partitions of the network (dynamic topology), we can notice that it is possible that all the data are not collected even if the reliability is maximal.

Parameter `aTimer`.

Figure 19 plots the average percentage of collected data as a function of Parameter `aTimer` according to five levels of reliability namely 0.1, 0.3, 0.5, 0.7 and 1. Parameter `maxloss` is fixed at 3. As depicted in Figure 19, our experiments shows that the shorter `aTimer` is, the greater the percentage of collected data is. The main reason stems from the fact that a short duration allows to terminate more data collections before a topology change occurs than a long duration.

Parameter `maxloss`.

To study the impact of `maxloss` on the number of collected data, for each $\texttt{maxloss} \in \{1,\ldots,10\}$, we run 50 simulations and we recorded the average percentage of collected data. Reliability and duration are respectively fixed at 1 and 2000 ms. As a result, when `maxloss` is fixed at 1, the initiator collects no data except its own identity: Indeed in this case and according to Algorithm 1, all the received data are always deleted because each time the timer expires, all the neighbors reach a lifetime of zero. On the other hand, when $\texttt{maxloss} \geq 2$ the average number of collected data remains relatively static. The main reason comes from the fact that the scenario is not dynamic enough and consequently a $\texttt{maxloss} = 2$ is sufficient to collect all the data in our scenario.

However, that is no longer the case through a less stable scenario allowing more dynamicity. Indeed, we tested the impact of `maxloss` via a scenario which consists of two convoys with opposite directions that repeatedly cross each other on a circular route. Due to their opposite directions, the two convoys are repeatedly disconnected and reconnected. As a result, the greater `maxloss` is, the greater the percentage of collected data is.

6.3 Discussion

Lab experiments highligh the effect of varying some parameters. So, it seems judicious to adapt these parameters according to the context and the desired type of collected data.

For instance, in a low dynamic network, it may be enough to fix `maxloss` at 2 because each neighborhood tends to remain relatively stable. On the other hand, in a very dynamic network, it seems appropriate to increase the value of `maxloss` so that to save as many data as possible from vehicles which have been in the neighborhood at least once. However, a very high `maxloss` prevents from maintaining the data up to date: This is a problem when the desired data are subject to fluctuations (e.g., speeds or positions). To circumvent the problem, timer duration `aTimer` may be reduced.

7. CONCLUSION

In this paper, we proposed a protocol which collects information using inter-vehicle communications only, *i.e.*, without using infrastructure or one of its components. The pro-

posed protocol is mainly based on a specific tool borrowed from the self-stabization area, namely Operator ant, which confers to our protocol the nice property of supporting the dynamicity. In particular, it is not required for vehicular networks to remain connected. We have implemented the proposed protocol and tested it via the Airplug middleware which allows field and lab experiments. So, road experiments confirmed that COL is able to support network disconnections contrary to existing protocols. Road experiments have been done with a convoy of 5 mobile vehicles in which they did not change place inside the convoy. However, the connections were dynamic due to experimental conditions. Via lab experiments, we have showed that COL still works even in the presence of frequent disconnections caused by dynamicity or a low level of reliability of communication links. We also highlighed how to partially circumvent the disconnections for instance by increasing the value of neighbor lifetime `maxloss` or by decreasing the value of the timer duration `aTimer`. As a future work, we would like consider COL as a basic protocol for designing more complex applications in vehicular networks. In particular, our protocol could be the basis of some kind of multi-criteria link-state routing protocol.

8. REFERENCES

[1] Airplug website. [Online]. Available: http://www.hds.utc.fr/airplug

[2] L. Bononi and M. D. Felice, "A cross layered mac and clustering scheme for efficient broadcast in vanets," in *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems*, 2007, pp. 1–8.

[3] O. Brukman, S. Dolev, Y. A. Haviv, and R. Yagel, "Self-stabilization as a foundation for autonomic computing," in *ARES*, 2007, pp. 991–998.

[4] A. Buisset, B. Ducourthial, F. E. Ali, and S. Khalfallah, "Vehicular networks emulation," in *19^{th} International Conference on Computer Communication Networks (ICCCN)*, August 2010.

[5] S. Capkun, M. Hamdi, and J.-P. Hubaux, "GPS-free positioning in mobile ad hoc networks," *Cluster Computing*, vol. 5, no. 2, pp. 157–167, 2002.

[6] S.-H. Chen and T.-L. Huang, "A wave algorithm for mobile ad hoc networks," in *Workshop on Algorithms and Computational Molecular Biology co-located with ICS*, 2002.

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

[8] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

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

[10] B. Ducourthial, "r-semi-groups: A generic approach for designing stabilizing silent tasks," in *9^{th} Stabilization, Safety, and Security of Distributed Systems (SSS'2007)*, November 2007, pp. 281–295.

[11] B. Ducourthial and F. Elali, "A light architecture for opportunistic vehicle-to-infrastructure communications," in *ACM International Symposium on Mobility Management and Wireless Access*, 2010.

[12] B. Ducourthial and S. Khalfallah, "A platform for road experiments," in *VTC Spring*, 2009.

[13] B. Ducourthial, S. Khalfallah, and F. Petit, "Best-effort group service in dynamic networks," in *SPAA*, 2010, pp. 233–242.

[14] B. Ducourthial and S. Tixeuil, "Self-stabilization with r-operators," *Distributed Computing*, vol. 14, no. 3, pp. 147–162, 2001.

[15] S. Finn, "Resynch procedures and a fail-safe network protocol," *IEEE Transactions on Communications*, vol. 27, no. 6, pp. 840–845, 1979.

[16] E. Karami and M. Shiva, "Maximum likelihood MIMO channel tracking," in *VTC*, 2004, pp. 876–879.

[17] S. Khalfallah and B. Ducourthial, "Bridging the gap between simulation and experimentation in vehicular networks," in *VTC Fall*, 2010, pp. 1–5.

[18] U. Lee and M. Gerla, "A survey of urban vehicular sensing platforms," *Computer Networks*, vol. 54, no. 4, pp. 527–544, 2010.

[19] U. Lee, E. Magistretti, B. Zhou, M. Gerla, P. Bellavista, and A. Corradi, "Efficient data harvesting in mobile sensor platforms," in *PerCom Workshops*, 2006, pp. 352–356.

[20] N. A. M. Efatmaneshnik, A. T. Balaei and A. Dempster, "A modified multidimensional scaling with embedded particle filter algorithm for cooperative positioning of vehicular networks," in *IEEE International Conference on Vehicular Electronics and Safety*, 2009.

[21] Z. Mo, H. Zhu, K. Makki, N. Pissinou, and M. Karimi, "On peer-to-peer location management in vehicular ad hoc networks," *International Journal of Interdisciplinary Telecommunications and Networking (IJITN)*, vol. 1, no. 2, pp. 28–45, 2009.

[22] T. Nadeem, S. Dashtinezhad, C. Liao, and L. Iftode, "Trafficview: traffic data dissemination using car-to-car communication," *Mobile Computing and Communications Review*, vol. 8, no. 3, pp. 6–19, 2004.

[23] R. Parker and S. Valaee, "Vehicular node localization using received-signal-strength indicator," in *IEEE Transactions on Vehicular Technology*, vol. 56, 2007, pp. 3371–3380.

[24] S. Roumeliotis and I. Rekleitis, "Propagation of uncertainty in cooperative multirobot localization: Analysis and experimental results," *Autonomous Robots*, vol. 17, no. 1, pp. 41–54, July 2004.

[25] I. Salhi, M. O. Cherif, and S.-M. Senouci, "A new architecture for data collection in vehicular networks," in *ICC*, 2009, pp. 1–6.

[26] A. Segall, “Distributed network protocols,” *IEEE Transactions on Information Theory*, vol. 29, no. 1, pp. 23–34, 1983.
[27] G. Tel, *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.
[28] H. Wu, R. M. Fujimoto, R. Guensler, and M. Hunter, “Mddv: a mobility-centric data dissemination algorithm for vehicular networks,” in *Vehicular Ad Hoc Networks*, 2004, pp. 47–56.