ALGORITHMS FOR STABLE CLUSTERING IN VANETS (VEHICULAR AD HOC NETWORKS)

Αλγόριθμοι για Σταθερή Ομαδοποίηση σε Δίκτυα Οχημάτων

Katsarou Foteini

Supervisors:
Dimitrios Katsaros
Leandros Tassiulas
Christos Antonopoulos

Department of Computer & Communication Engineering

University of Thessaly

Volos, September 2012

Πανεπιστήμιο Θεσσαλίας

Τμήμα Μηχανικών Η/Υ τηλεπικοινωνιών & Δικτύων

Μεταπτυχιακή Διπλωματική Εργασία

ΑΛΓΟΡΙΘΜΟΙ ΓΙΑ ΣΤΑΘΕΡΗ ΟΜΑΔΟΠΟΙΗΣΗ ΣΕ ΔΙΚΤΥΑ ΟΧΗΜΑΤΩΝ

Επιμέλεια:

Κατσαρού Φωτεινή

Επιβλέποντες:

Δημήτριος Κατσαρός

Λέανδρος Τασιούλας

Χρήστος Αντωνόπουλος

Acknowledgments

With the completion of my studies on the Department of Computer and Communication Engineering I would like to thank everyone who helped me in an immediate or a circular way for the achievement of this goal.

First of all, I would like to thank Mr. Dimitrio Katsaro, Leandro Tassiula and Christo Antonopoulo for all their useful and helpful advices and for the time they disposed. I would especially like to thank Mr. Dimitrio Katsaro and Christo Antonopoulo who enabled me to be a teaching assistant, parallel to my postgraduate studies, on the labs of the lessons they instructed during the academic year 2011-2012 Programming language I (C) and II (C++) correspondingly. From this opportunity, I gained more experience on both languages and through the teaching process I had the chance to test my capabilities and I received a deeper understanding of some more advanced concepts. Additionally I would like to thank my fellow-student Pavlo Bassara for his time and all the useful tips concerning the programs SUMO and NS3.

Additionally, I want to thank my friends and the people who existed near me and supported me during my studies, from the first time I attended the department for my undergraduate studies till now that I receive my Master. Altogether, we shared both beautiful moments and concerns too. I feel very lucky that they exist in my life and without them I might never have the chance to reach at this point. At this point I would really like to thank a person especially important for me, George, who stands by my side with a lot of patience and care.

Finally, I would like to thank my family, who stands at my side in every choice I make, who supports and guides me. The present thesis is dedicated to my family who taught me above all to lean on my feet and to fight for my dreams.

Abstract

VANETs (Vehicular Ad Hoc Networks) are a set of vehicles (nodes) equipped with some antenna that are moving constantly by following the direction of a road and by forming platoons, especially when moving on highways. The peculiarity of their movement brought the necessity for efficient algorithms that would facilitate the communication of such nodes.

Moreover, in order to support complex applications such as safety warnings in case of an accident, broadcast of comfort applications such as music file exchange the users, broadcast of information concerning traffic jams and traffic management, it is required to achieve stable clustering among the nodes. Clustering refers to the division of nodes in virtual groups and they are allocated in such a way that they are geographically adjacent into the same cluster. Nodes within the same cluster comply to the rules of the cluster, which makes them present different behavior from the nodes outside the cluster.

Nodes within a cluster are classified in 3 major categories: clusterheads, gateways and ordinal nodes or member nodes. Nodes that do not belong in any cluster are orphan nodes or NULL state nodes.

In the following pages, there are presented 3 different algorithms for clustering in VANETs. Firstly, it is presented the algorithms and the code files that were produced in order to implement them. On the following, it is presented a comparison among the 3 algorithms related to the average number of messages that are exchanged among the nodes, the average number of clusters that are produced and the average size of clusters.

Chapter 1 - Introduction

The main topic of this thesis is to study the capability of existing algorithms for supporting Stable Clustering on VANETs. VANETs stand for Vehicular Ad Hoc Networks and the need for Stable Clustering emerged due to the requirements of efficient communication between the nodes without flooding the whole network with unnecessary and redundant messages. Clustering of nodes provides an efficient means of establishing a hierarchical structure in mobile ad hoc networks (MANETs).

We have to mention that VANETs are a subcategory of MANETs because of the specific characteristics of VANETs. So, the algorithms proposed for MANETs are not totally applied on VANETs, or they may even be completely inappropriate.

In the chapters that follow, it is provided explanation of the basic terminology that will be used on this thesis, it follows the algorithms that were implemented and finally the comparison between them.

More specifically, Chapter 2 provides a quick introduction to ad hoc networks and their basic characteristics. It follows the difference between MANETs and VANETs and the possible application of VANETs in realistic problems. Then, we introduce the basics of clustering and the categorization of nodes during this process along with the benefits in the communication process. Additionally, we indicate a quick introduction to the algorithms that were implemented on this thesis. Finally on this chapter, it exists a reference of the basic techniques that can be used for the position retrieval information of a vehicle.

Chapter 3 refers to the creation of the simulation network. Firstly we provide the details for the programs that were used to create the network along with some installation guides and tips for the running process. In more detail, we used SUMO to produce some traffic mobility and C++ code for the simulation. In order to achieve the export of SUMO files in trace files, that are more easily conceivable by the C++ code that was implemented, it was used TraceExporter, which is a SUMO extension. The files that are produced by this method are also conceivable by NS2. It follows the presentation of the created topology experiments for our simulation.

The following chapter (Chapter 4) is dedicated to the algorithms that were implemented for this thesis. The main pattern that is used for this chapter is a quick reference to the basic characteristics of each algorithm, the presentation of some implementation remarks and finally the presentation of the code files that were produced.

Chapter 5 is the conclusion of this thesis and it summarizes all the previous algorithms by comparing their main characteristics. Each algorithm is evaluated against the others concerning the average number of messages that are exchanged among the nodes, the average number of clusters that are produced and the average size of clusters.

Chapter 2 – Fundamentals of VANETs

In this chapter we will discuss the basic terminology around VANETs, before we move on to the presentation of the basic algorithms and techniques for clustering. The presented concepts are those of ad hoc networks, the differences between VANETs and MANETs and their possible applications. It follows the necessity for clustering and how they are related to VANETs. Finally, there are presented sententiously the basic techniques for retrieving the vehicle's position.

2.1) Introduction to Ad Hoc Networks

The term **ad hoc network** refers to **wireless** networks that are decentralized (Figure 1). The basic characteristic is that they do **not** rely on a **preexisting infrastructure**. Ad hoc network refers to a set of devices that have equal status on the network and are free to associate with any other network device in link range. During the communication process between nodes, the participants of the same network usually need to create a **backbone** of the network before they try to exchange messages with each other. A famous technique of creating a backbone of the network is **clustering**. Alternatively, the communication could be efficient through an excessive exchange of broadcast messages (**flooding**). But this technique would flood the network with redundant messages and would lead to contention and collision problems. Clustering aims to the creation of groups of nodes in such a way that they can communicate with each other with the minimum possible exchange of redundant messages. But before we proceed, we will analyze what decentralization means, and what are the advantages of this technique in ad hoc networks.

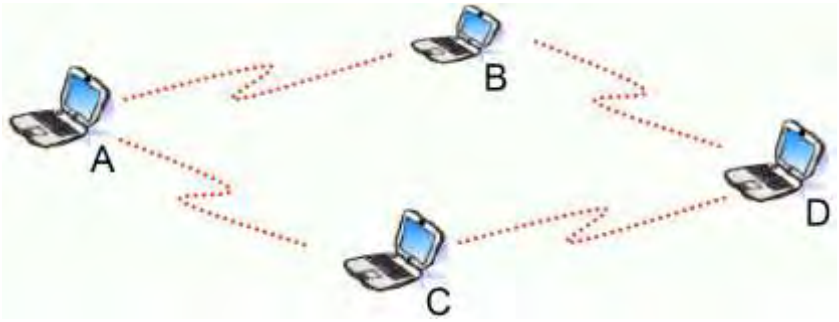

Figure 1: An example of ad hoc network

Networks can be divided into two major categories concerning their **organizational structure**. They can be either centralized or decentralized. A **centralized** network is a network where the activities regarding planning and decision-making concerning clustering or other issues, become concentrated within a particular node and/or a group of nodes. Of course, this means that those nodes are highly aware of the exact topology of the rest of the network. As opposed to centralized networks, **decentralized** networks are characterized by the dispersion of the decision-making to practically all the nodes of the network. It can easily be comprehended that the centralized version is totally inappropriate for ad hoc networks, mostly because of the non-preexisting infrastructure. So it's impossible for a specific node to know the exact position of all other nodes, even worse when we take into account mobility of nodes.

The **decentralized nature** of ad-hoc networks makes them suitable for a variety of **applications** where central nodes can't be relied on. First of all, we have to take into account the scalability of wireless ad-hoc networks compared to wireless managed networks. Minimal configuration and quick deployment make ad hoc networks suitable for emergency situations like natural disasters or military conflicts and in places where the network infrastructure is not taken for granted. The presence of dynamic and adaptive routing

protocols enables ad-hoc networks to be formed quickly. Although, there have been identified theoretical and practical limits to the overall capacity of such networks, that will be discussed later.

Wireless ad hoc networks can be further classified by their application to the following categories:

- **mobile ad-hoc networks** (MANET), is a self-configuring infrastructure-less network of mobile devices connected wirelessly. They will be discussed in more detail later.
- **wireless mesh networks** (WMN), which are communication networks made up of radio nodes organized in a mesh topology. Wireless mesh networks often consist of mesh clients, mesh routers and gateways. The mesh clients are often laptops, cell phones and other wireless devices while the mesh routers forward traffic to and from the gateways which may not need to connect to the Internet. A wireless mesh network often has a more planned configuration than the ad hoc network, and may be deployed to provide dynamic and cost effective connectivity over a certain geographic area. Whereas, an ad-hoc network is formed ad hoc when wireless devices come within communication range of each other.
- **wireless sensor networks** (WSN), which consist of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, pressure, ... and to cooperatively pass their data through the network to a main location. The more modern networks are bi-directional, by also enabling control of sensor activity.

An ad-hoc network is made up of multiple nodes connected by "**links**". The links are not wired and their achievement can't be taken for granted. There are many **practical and theoretical limits** that affect the achievement of links. Firstly, they are influenced by the **node's resources** (e.g. transmitter power, computing power and memory) and by **behavioral properties** (e.g. reliability), as well as by **link properties** (e.g. length-of-link and signal loss, interference and noise). Furthermore, links can be connected or disconnected at any time, mostly because of the mobility of both the nodes or because of the emergence of obstacles on the current environment of the network. So, a functioning network must be able to cope with this **dynamic restructuring**, preferably in a way that is timely, efficient, reliable, robust and scalable. Another practical issue in most wireless ad hoc networks, is that the nodes **compete** for **access to the shared wireless medium**, which often results in collisions (**interference**). The use of cooperative wireless communications[1] improves the immunity to interference by having the destination node combine self-interference and other-nodes interference to improve decoding of the desired signal.

As far as it concerns the **dynamic restructuring**, the network must allow any two nodes of the network to communicate, by relaying the information via other nodes. This includes the creation of a (certain) "**path**", which is a series of links that connects two nodes. Various routing methods use one or two paths between any two nodes; flooding methods use all or most of the available paths. When talking about **broadcasting**, where a message has for recipients all the other nodes, flooding is not the best option as discussed before. Instead of that, clustering is preferred.

2.2) MANETs and VANETs

Previously on this chapter, it was mentioned the term **MANET**, which stands for **mobile ad-hoc network** and is a set of mobile devices connected wirelessly forming a network with no certain infrastructure but able of being self-configured. The devices are allowed to move freely in any direction, which is the main

[1] "Cooperative Wireless Communications: also known as multi-user MIMO (MU-MIMO) is a set of advanced MIMO, multiple-input and multiple-output, technologies that exploit the availability of multiple independent radio terminals in order to enhance the communication capabilities of each individual terminal. To contrast, single-user MIMO only considers access to the multiple antennas that are physically connected to each individual terminal. MU-MIMO can be seen as the extended concept of space-division multiple access (SDMA) which allows a terminal to transmit (or receive) signal to (or from) multiple users in the same band simultaneously." (Wikipedia)

reason for the frequent change of the links between them. Furthermore, every device acts like a router, by forwarding traffic unrelated to its own use. So each device must be equipped with a mechanism of maintaining and updating such information in order to properly route traffic. Such networks may operate by themselves or may be connected to the larger Internet. Another thing we have to notice about MANETs is that they may lead to frequent disconnection of the network, especially when vehicle density is low. The high mobility of such nodes could also lead to hard delay constraints in the transmission of a message.

There are many different algorithms proposed for MANETs in order to augment the stability of the routing infrastructure and to minimize the exchanged messages between the nodes and the packet drops due to bad routing decisions, etc.

VANETs is a certain type of MANETs. The acronym VANET stands for **Vehicular Ad Hoc Network** and **resembles** to MANET because they both **change rapidly** and **dynamically**. VANETs don't have a pre-established infrastructure and they lack of central control. Also, devices on both MANETs and VANETs move freely in any direction, but in **VANETs** we have the extra limitation of movement within **predefined roads**. Additionally, in VANETs, the **speeds of the vehicles** are bounded due to speed limits, level of congestion and traffic control mechanisms (e.g. traffic lights and "stop" signs on the roads). Last but not least, future vehicles could be equipped with much more capabilities such as antennas with longer transmission ranges, extensive on board storage capacities, more processing power and rechargeable sources of energy which can be achieved easily in VANETs but not in MANETs. On Table 1 that follows, it is presented the comparative main characteristics of VANETs and MANETs which was originally cited on the paper titled "A Comparative study of MANET and VANET Environment" by Arzoo Dahiya , Dr.R.K.Chauhan.

Parameter	MANET	VANET
Cost of production	Cheap	Expensive
Change of the network topology	Slow	Frequent and very fast
Mobility	Low	High
Node density	Sparse	Dense and frequently variable
Bandwidth	Hundred kps	Thousand kps
Range	Up to 100m	Up to 500m
Node lifetime	Depends on power resource	Depends on lifetime of vehicle
Moving pattern on nodes	Random	Regular
Position acquisition	Using ultrasonic	Using GPS, RADAR

Table 1: Summary of the basic characteristic of MANETs and VANETs, as presented on "A comparative study of MANET and VANET Environment".

2.3) Possible applications on VANETs

There do exist a number of attractive applications for VANETs. For begging, we have to mention that due to the similarity between VANETs and MANETs, many applications of MANETs could be used for VANETs too, but the details differ. Instead of the random movement of vehicles, on VANETs, they have to proceed in an organized way and restricted by roadsides. The interactions with roadside equipment can likewise be characterized fairly accurately.

One of the most essential application is the provision of driving safety through the assistance of drivers in the avoidance of collisions and the supply of an alternative route among highway entries and intersections. This could be easily implemented in accordance to GPS and navigation systems. Additionally, the

driver could be supplied with traffic jam reports, high-speed tolling and the fastest routes to the work. Future applications could involve cruise control making automatic adjustments to maintain safe distances between vehicles or alerting the driver of emergency vehicles in the area.

Another very useful application is the capability of transforming a traffic jam into a productive work time, by having the e-mails downloaded and read. The system could also allow VoIP services such as Google-Talk or Skype between employees.

Finally, Intervehicular Communications (IVC) could be used to provide comfort applications such as file exchange, interactive communications, weather information, gas station or restaurant locations and their vicinity to the requester. Other than IVC, we could also have RVC (Road Vehicle Communications) applications.

All in all, the possible applications vary between traffic management, road monitoring, entertainment and contextual information.

2.3) Clustering

As referred previously, VANETs are **infrastructure-less** networks and the **exact topology** of the network **isn't known** to every node of the network. So, in order to avoid flooding, we have to create a **backbone** of the network before vehicles start exchanging messages. In our case, we want to create an abstract structure over the network, so that regional changes doesn't need to be made perceivable from all the network. This, can be achieved by the use of clusters.

Clustering is the process of creating such substructures on the whole topology network. A more formal definition given in the paper titled "A Survey of Clustering Schemes for Mobile Ad Hoc Networks" is the following: "In a clustering scheme the mobile nodes are divided into different virtual groups and they are allocated geographically adjacent into the same cluster according to some rules with different behavior for nodes included in a cluster from those excluded from the cluster". All the nodes are classified to one of the following 3 categories which is also shown in the Figure 2:

1. **clusterheads** of the cluster: they have to rebroadcast the message to every member of the group and they act as the local coordinators for their cluster,
2. **gateways**: they rebroadcast the message to their neighboring heads of the clusters and they are used for communication between clusters,
3. **ordinal nodes (or cluster members)**: they just receive the message within the cluster without rebroadcasting it to anyone else.

Clusterheads and gateways form the backbone of the network. Gateways may belong to more than one clusters. In order to apply some algorithm scheme, each node of the network receives a unique identifier (ID). We have to mention that during the process of clustering in a highly dynamic network, we may have some **orphan nodes** which are mainly ex-members of a cluster that currently don't belong to any cluster.

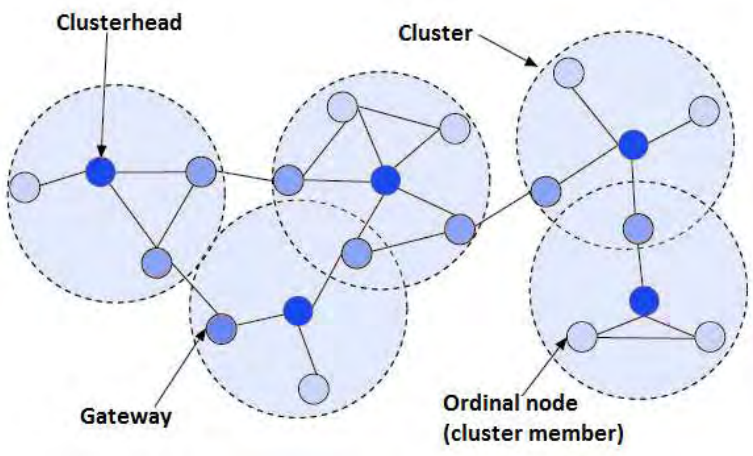

Figure 2: Clustering, the fundamentals

The **basic advantages** of clustering can be summarized to 1) the spatial reuse of resources, where on a non-overlapping multicast structure, two clusters may use the same frequency or code if they don't belong to neighboring clusters, 2) the use of a virtual backbone, formed by clusterheads and gateways, in order to avoid flooding and 3) the improved network stability and scalability as far as it concerns a regular member where local changes need not to be seen and updated by the entire network.

Additionally to the advantages of clustering we have to take into account the **cost of creating and maintaining** clusters in such networks. Firstly, we have to mention that in order to maintain clustering structures, nodes in some pairs have to exchange messages periodically. The period time for these messages to be exchanged differs concerning the mobility patterns of the nodes. But in a rapidly changeable network, there have to be exchanged enough messages which leads to energy and bandwidth consumption. Secondly, some cluster schemes may cause the cluster structure to be completely rebuilt over the whole network when some local events take place such as the "death" of a node (ripple effect or re-clustering). Thirdly, many clustering schemes work on two phases: the phase of cluster formation and the phase of cluster maintenance. Some of them assume that during the cluster formation the nodes maintain their positions which is not at all accurate in realistic scenarios. In this case we have also to consider the time required for the cluster formation which is called *computation round*. This time may varies from algorithm to algorithm and the nodes can' t stay steel for a long time.

There are many **algorithms**, proposed for clustering in MANETs and VANETs. We have to mention that MANETs protocols aren't suitable for VANETs. The reason is that protocols designed for MANETs don't take into account the unique characteristics of VANETs concerning the movement patterns, so they are not suitable for intervehicular communication. **MANETs routing protocols** can be classified in 3 major categories:

a) **proactive**: that maintain and update information on routing between all nodes of a given network at all times,
b) **reactive**: where the route determination is invoked on a demand or on need basis,
c) **hybrid**: these protocols are a mixture of both proactive and reactive protocols. A characteristic example is zone routing protocol.

In our case, the more suitable are the reactive algorithms, in order to diminish flooding messages. But reactive algorithms don't take into account mobility parameters such as break of links. They also cause flooding during the route discovery process.

Another classification for MANETs algorithms, concerning the dominant role of the clusterhead on the formation and maintenance of the cluster is the following:

a) **Lowest-ID** algorithm: where the nodes periodically broadcast the list of nodes that can hear including its own and the node becomes clusterhead if in its neighbor he has the lower ID.
b) **Highest-Degree** algorithm: where the degree of a node is computed based on its distance from others. The node with the maximum number of neighbor (e.g. maximum degree) becomes clusterhead.
c) **Beacon-Based** algorithm: which is based on the periodical/regular transmission of hello messages which advertise the state of a node. Based on the state of the neighbors, a node can select its own state. A clusterhead will consider a change of its state if it receives a message from another clusterhead. A clusterhead receiving a hello message from another CH will remain in the same state if it has more cluster members of its own than the sender.

A beacon-based algorithm is the only applicable kind of algorithm on VANETs.

2.4) Platooning on VANETs

Platooning is the tendency of vehicles to move relatively close together, especially in urban environments or on highways, by forming groups with similar velocities and direction of motion, thus creating topologies that look like platoons. These platoons of vehicles frequently pass one another in opposite directions or mingle on traffic lights.

Platooning is especially useful in VANETs' algorithms, as they profit from the fact that during cluster formation and in order to maintain stability and relative resilience, the nodes give preference for groups which increase the node cohesion, e.g. nodes that travel towards the same direction and with similar velocities.

2.5) Algorithms that were implemented

A clustering algorithm must strive to maintain cluster stability and retain the cluster contents and structure for as long as possible, as otherwise, the frequent re-clustering processes will degrade the performance of the communication.

In the following pages they will be presented in detail 5 algorithms. The first algorithm is called APROVE (Affinity PROpagation for Vehicular networks) and was presented on the paper titled "Mobility-Based Clustering in VANETs using Affinity Propagation" by Christine Shea, Behnam Hassanabadi and Shahrokh Valaee. The main idea is the utilization of the Affinity Propagation in a distributed manner. Each node in the network transmits the responsibility and availability messages to its neighbors and then makes a decision on clustering independently.

The second algorithm is an aggregate local mobility clustering algorithm and was presented on the paper "A New Aggregate Local Mobility (ALM) Clustering Algorithm for VANETs" by Evandro Souza, Ioannis Nikolaidis and Pawel Giburzynski. The algorithm's philosophy is the use of RSS as an indication of the distance between the sender and the receiver but clusterheads exchange more than one packet within a certain amount of time to avoid break leakage and the node's decision regarding its status change is based on its perception of the aggregate local mobility (ALM).

The third algorithm named "Stable Clustering and Communications in Pseudolinear Highly Mobile Ad Hoc Networks" by Ehssan Sakhaee and Abbas Jamalipour refers to a primary MANET algorithm, which is applicable in VANETs. There are presented two algorithms concerning the available position information of the

nodes both of which consider two phases. The first phase is cluster formation (initial clustering) and the second one is cluster maintenance (progressive). It will be implemented only the algorithm related to retrieval of a node's position through GPS.

2.6) Techniques for retrieving position

Before we proceed, we have to mention that all algorithms require to know some position information in order to function properly. Each vehicle needs either an efficient way for determining their absolute position in the bounds of a city / road frame or its relative position towards the other vehicles. The most famous techniques to do so, are:

- **GPS** (Global Positioning System)
- Doppler effect from where we export the **Doppler Value**
- **RSS** (Received Signal Strength)

On the following algorithms, the technique that was used to retrieve position information is GPS.

Chapter 3 – Creating the simulation network

Before the simulation of the algorithms, it emerged the problem of simulating an architecture of the network. In more depth, we had to create a realistic environment where the vehicles could move with some velocity towards a random direction but into predetermined roads. The development of such an architecture is based on the paper "Ad Hoc Peer-to-Peer Network Architecture for Vehicle Safety Communications", written by Wai Chen and Shengwei Cai.

The simulation of the networks was made with SUMO. In the following pages, there is a detailed reference of the tool (SUMO) and the steps that were followed for the installation, along with some quick example to demonstrate its usage. It will then follow the presentation of the created environment for the simulation of the algorithms.

3.1) Introduction to SUMO

3.1.1) What is SUMO

SUMO stands for **Simulation of Urban MObility** and is licensed under the GPL (GNU General Public License). It is **open source** and its use is related to the creation and simulation of highly portable, microscopic and continuous road traffic packages which are designed to handle large road networks. In more detail a network file that is produced with SUMO is capable of describing the traffic related to a part of a map. It contains the roads of the network with intersections or junctions along with the traffic lights of the map.

The **format** of the files is **XML** (eXtensible Markup Language), but the files that are produced are not meant to be edited by hand. There are some programs (NETGEN, NETCONVERT) which are related to SUMO and allow the creation of networks in a very comfortable way. Also, SUMO offers a wide variety of extensions in order to **communicate** thriftily with other programs, such as **ns3**, which is necessary for our simulation.

An **alternative program** for generating movement traces is **VanetMobiSim** which is an extension for the CANU Mobility Simulation Environment (CanuMobiSim). CanuMobiSim is JAVA-based and can generate movement traces in different formats, supporting different simulation/emulation tools for mobile networks (NS2, GloMoSim, QualNet, NET).

3.1.2) The format of the network

In an abstraction manner, we can say that a SUMO network is a directional graph where the edges are the roads and the nodes are the intersections/junctions. In more depth, further traffic is related to the following information:

- every street/road is a collection of lanes,
- we can define a certain position, shape and speed limit on every lane,
- the right-of-way regulation,
- the connections between lanes (that correspond to junctions/intersections),
- the position and the logic of the traffic lights.

3.1.3) Installation procedure

Here is presented the basic procedure for installing SUMO on Ubuntu 11.04 (Natty Narwhal). The full installation guide for other systems too can be found on the following link:
http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Installing/Linux_Build

The version that was installed is 0.15.0. The basic installation steps are:

→ Open a terminal

→ SUMO needs some other tools too, such as Proj, GDAL, Fox (for GUI) and Xerces-C (if it doesn't come with the distribution).
sudo add-apt-repository ppa:gwibber-daily/ppa
With the previous command, we declare on Ubuntu where to find the PPA (Personal Package Archive).
Now, we have to tell the system to pull down the latest list of software from each archive it knows about, including the PPA we just added:
sudo apt-get update
And we are ready to start installing software from the PPA.
sudo add-apt-repository ppa:sumo/stable
sudo apt-get install sumo sumo-tools sumo-doc
We have to provide root access for some steps. So, we use the following command, which requires the root's password:
su root
The following packages are needed to build SUMO:
sudo apt-get install libtool libgdal1-dev proj libxerces-c2-dev
sudo apt-get install libfox-1.6-dev libgl1-mesa-dev libglu1-mesa-dev
cd /usr/lib; sudo ln -s libgdal1.6.0.so libgdal.so
(Note: We can safely ignore gdal failure due to Python bindings.)

→ Now, we need to get source code, which we can download on the link:
http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Downloads
The name of the file we want is *sumo-src-0.15.0.tar.gz*.
Afterwards, we have to go to the folder that the tar is stored.
tar xzf sumo-src-0.15.0.tar.gz
cd sumo-0.15.0/

→ It follows the building of the SUMO binaries:
./configure --with-fox-includes=/usr/include/fox-1.6
--with-gdal-includes=/usr/include/gdal --with-proj-libraries=/usr
--with-gdal-libraries=/usr --with-proj-gdal
make

→ The final step of installing the SUMO binaries is optional. This will copy the SUMO binaries to another path, so that we can delete all source and intermediate files afterwards. If you do not want to do that, we can simply skip this step and run SUMO from the source subfolder (src/sumo-gui and src/sumo).
make install

3.2) Introduction to NS

3.2.1) What is NS

NS3 is a discrete-event network simulator and is mainly used in the simulation of ad hoc networks. Previous versions are ns-1 and ns-2 and the current version is ns-3. It is targeted for research and educational use.

The development of ns-3 project started in 2006 and it is open-source. It is implemented in C++ and Python and it is licensed under the GNU GPL (General Public License). The workflow on ns includes the following 4 steps:

1. Implementation of protocol models,
2. Setup the simulation scenario,
3. Running the simulation,
4. Analysis of the simulation results

Some basic components of the program are:

1. NS, the simulator itself has Nam, the network animator to visualize ns (or other) output,
2. Pre-processing component for Traffic and topology generators,
3. Post-processing for Simple trace analysis (in Awk, Perl or Tcl).

3.2.2) Producing traffic mobility for the NS2 simulation

A solution is to use SUMO in order to produce the traffic and then use the TraceExporter (SUMO's extension) to export SUMO traces. TraceExporter is written in Java and by its usage there are produced three files: config, activity and mobility. It follows the basic steps to achieve this.

→ Let's assume that under a certain folder we have "hello.nod.xml" and "hello.edg.xml" files, that can produce the "hello.net.xml". (We might need to add the "hello.typ.xml" file in the creation of the "hello.net.xml" if we have one such file.) Then the files "hello.net.xml" and "hello.rou.xml" will produce the "netstate.xml" by calling the sumo execution. The following 2 commands, outline the procedure:

```
# netconvert --node-files=hello.nod.xml --edge-files=hello.edg.xml  --type-files=hello.typ.xml --output-file=hello.net.xml
# sumo-gui -n hello.net.xml -r hello.rou.xml --netstate-dump netstate.xml
```

→ After having downloaded SUMO, in the relative path "tools/traceExporter", we can find the file "traceExporter.jar". We have to copy this file in the folder that exist the previously mentioned files. Then in the terminal we can use the following command:

```
#java -jar traceExporter.jar ns2 -n hello.net.xml -t netstate.xml -a activity.tcl -m mobility.tcl -c config.tcl -p 1 -b 0 -e 150
```

where:

-n <net_file>
-t <trace_file>
-a <activity.tcl>
-m <mobility.tcl>
-c <config_file>
-p <penetration_level> (Penetration level means the ratio of selecting vehicles that will be traced for output and is a float in [0, 1]. Penetration level of 1 equals to no filtering and 0 means no vehicles will be selected.)
-b begin_time
-e end_time

→ The previous command will generate three files under the folder with names "activity.tcl", "mobility.tcl" and "con-

fig.tcl".

- The file "activity.tcl" describes the time that each vehicle does its first and last movement.
- The file "config.tcl" is used to set some opt()-variables which describe the simulation scenario.
- The file "mobility.tcl" contain the actual movements of the nodes. This is the file that is necessary for ns-3 and for our C++ code. We have to move this file under the folder where our C++ code is placed and rename it to "default.ns_movements".

Note that some basic characteristics of the TraceExporter are presented in the link: http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Tools/TraceExporter

3.3) The Simulation Networks

Having explored the basic components of both SUMO and NS3, we can move on to the presentation of the topologies of the 2 networks that were used for the simulation.

The first topology simulates the movement of 100 nodes in a highway. The topology consists of 2 roads (edges) that are moving in a parallel way and towards the same direction. The first road consists of 3 lanes and the second one consists of 2 lanes. The topology can be viewed at the Figure 3. During the simulation, they were constructed 100 nodes – vehicles of 6 different types (carA to carF). The vehicles varied slightly in the maximum velocity that they could move and the minimum gap that represents the minimum distance that they have to keep from the car that moves in the front. The details about the movement of the vehicles were declared in the "simple.rou.xml" file.

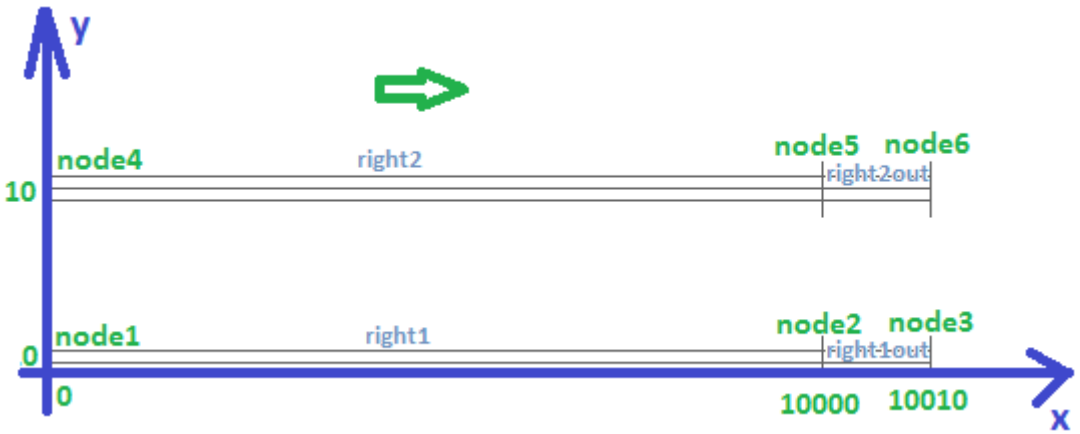

Figure 3: Representation of the highway (simple road) that was used for the comparison of the algorithms. Node1-6 represent the nodes that were defined on the "simple.nod.xml" file. Right1, right1out, right2 and right2out represent the edges that were defined on the "simple.edg.xml" file. They were used 2 different types of roads. The first type consists of 2 lanes and the second type consists of 3 lanes. Finally the nodes are moving towards the right direction.

The second topology consists of 12x12 roads (edges) that forms squares. The 12 horizontal edges are intersected with the 12 vertical edges forming 144 possible intersections (nodes) where a node could change its moving direction. The topology can be viewed at the Figure 4. Once again, during the simulation, they were constructed 100 nodes – vehicles of 6 different types (carA to carF). The vehicles varied slightly in the maximum velocity that they could move and the minimum gap that represents the minimum distance that they have to keep from the car that moves in the front. The details about the movement of the vehicles were declared in the "simple.rou.xml" file. In the same file, they were declared 50 possible routes that a vehicle could follow. Every 2 nodes followed the exact same route.

Here, we have to notice that in order to produce platoons on some roads, some edges were constantly preferred among others. This choice could simulate some very jammed roads on rush hours. It would be interesting to add even more vehicles in the topology, in order to produce fully crammed roads. In this scenario, it is quite easy to produce vehicles that cannot communicate with any other node at all, cause the transmission radius is low enough. In fact, in a real city transmission radius is even smaller than in an urban area or a highway, because of the obstacles that emerge from the buildings of a city.

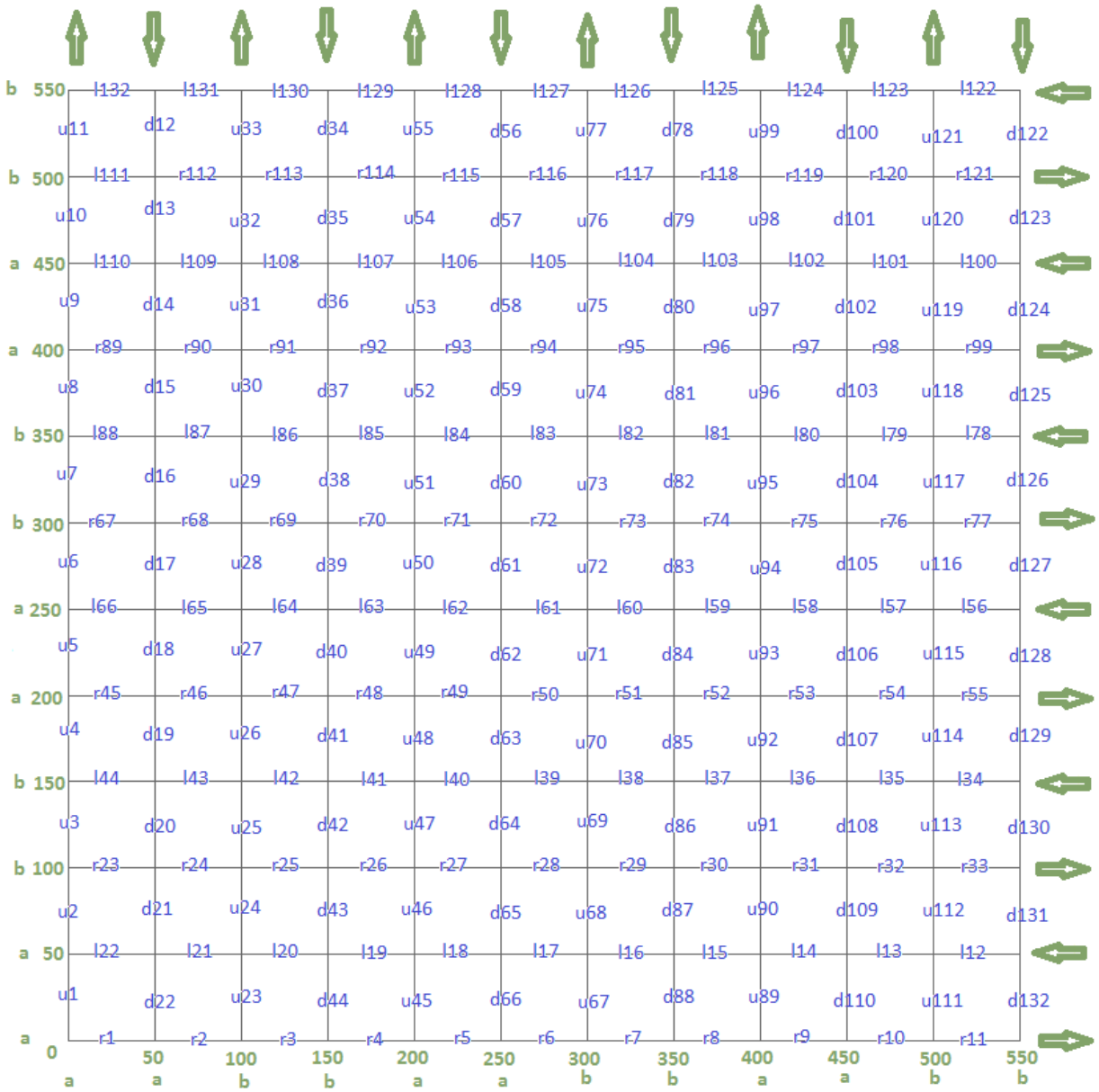

Figure 4: Representation of the city topology that was used for the comparison of the algorithms. They were declared 144 nodes on the "city.nod.xml" file. (In order not to make the scheme more complicated, the numbering of the nodes was omitted.) r#, l#, u# and d# represent the edges that were defined on the "simple.edg.xml" file (r for right, l for left, u for up, d for down also specify the allowed direction of the vehicles at each edge - road). They were used 2 different types of roads. Both types consist of 1 lane. So, the presented scheme consists of 144 intersections.

3.4) Initial files and description

In order to facilitate the code implementation for all the algorithms, they were produced some initial files that read the network which is ready for simulation and that load on the memory the position details of each node. The files that are almost identical for every algorithm are:

- **zzz_main.cc**: which contains the call of the main function. During the execution of the program, it is designed to accept at least 3 arguments with a certain order: (1) number of nodes that participate in the simulation, (2) the period of times that the simulation lasts and (3) the name of the file that contains the simulation details and has been produced with the TraceExporter (the SUMO's extension that was described previously). If the user doesn't provide the adequate number of arguments, the program is terminated by providing to the user the message *"Please enter the number of nodes and the period of times that the program runs as the second and the third parameter. Also give the filename that declares the movement."*. In the end of the file, there is a call to the main function of each of the three algorithms that were implemented. Finally, this file sets the transmission radio of the nodes that participate in the simulation.
- **movement.h** & **movement.cc**: These 2 files implement the class "Movement", which contains details read from the input simulation file that is declared by the user. Each node keeps its nodeID, its position (int x, int y) and its velocity (which is the total velocity on x and y axes and is not really necessary for the implemented algorithms).

Here, we have to note that if there are no details for some certain node for a certain period of time, it is used the position that the node had on the last-mentioned period time (simply in order to avoid undefined (not-set) positions).

Moreover, whenever necessary, velocity on each axel is calculated as presented on Eq. 1:

$$v_x = \frac{pos\_x_t - pos\_x_{t-1}}{t} \overset{t=1}{\Longrightarrow} pos\_x_t - pos\_x_{t-1}$$
$$v_y = \frac{pos\_y_t - pos\_y_{t-1}}{t} \overset{t=1}{\Longrightarrow} pos\_y_t - pos\_y_{t-1} \quad \text{Eq. 1}$$

Chapter 4 – Algorithms for Stable Clustering in VANETs

This Chapter is dedicated to the presentation of the 3 algorithms that were mentioned before. Firstly, it is presented the algorithm with the basic procedures, equations and figures. Then, there are presented some notes above the algorithm that refer to the its implementation. On the following it is presented the structure of the code files and their linking (with #includes).

4.1) Why is it important to consider only nodes that are moving towards the same direction?

To demonstrate the advantage of grouping vehicles, it is presented the example on Figure 5.

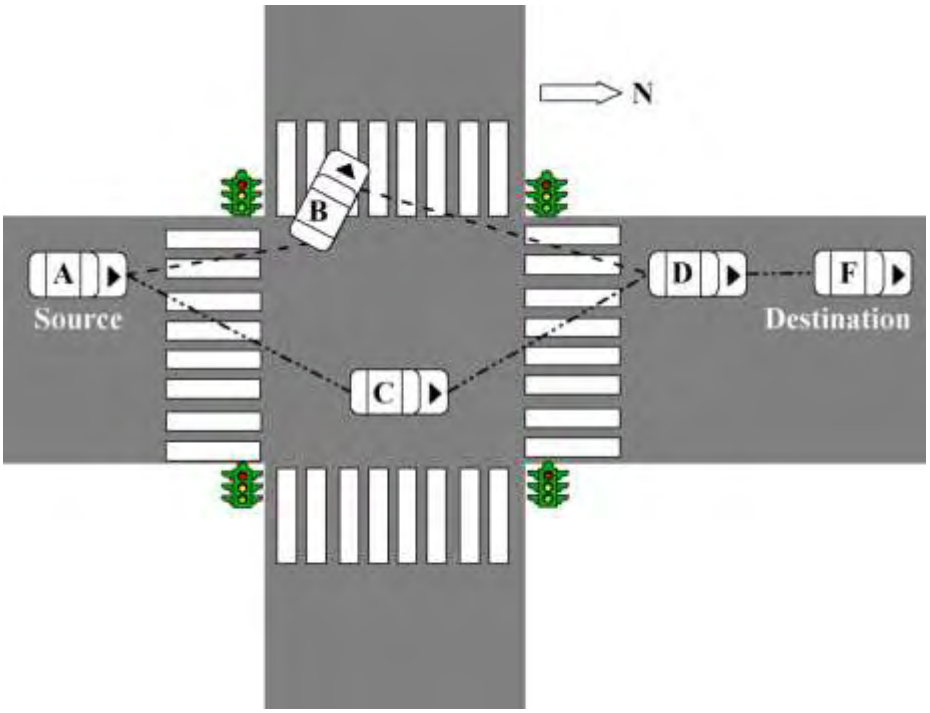

Figure 5: Link rupture event is more likely to occur between vehicles A, B and D. Analytically, 5 vehicles are moving towards an intersection and vehicle B is turning onto a new street, whereas the other 4 vehicles are continuing straight on the same road. A connection is established between vehicles A and F. Communication is possible on 2 routes: one via vehicle B (route A-B-D-F) and the other via vehicle C (route A-C-D-F). As vehicle B is turning left and vehicle A is continuing straight, the former route is more likely to be ruptured after a certain time. Consequently, the selection of the second route is more suitable and adds more stability and reliability to the communication path between the 2 vehicles (A and F).

In the proposed routing scheme, vehicles are grouped into four different groups based on their velocity vectors. In a Cartesian space, each vehicle is characterized by one of the unit vectors *[S_1 = (1,0), S_2 =(0,1), S_3 = (-1,0), S_4 = (0,-1)]*, as shown in Figure 6. Vehicles are assumed to be equipped with GPS device s to detect their geographical location. Location detection is performed every 1s time interval. Let $V_A = (v_x, v_y)$ denote the Cartesian coordinates of the velocity vector of a given vehicle A. By using the velocity and unit vectors, the group of vehicle A can be decided as follows. For example, vehicle A belongs to the group N if the dot product of its velocity vector and the unit vector S_N $[(V_A * S_N)]$ takes the maximum value for $N = 1$ on Figure 5.

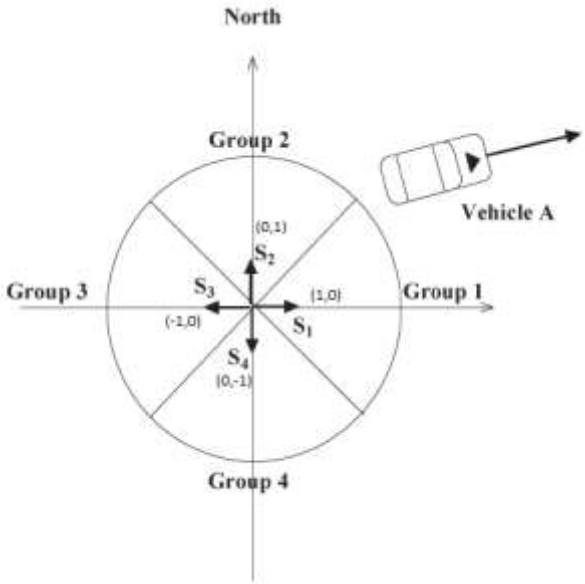

Figure 6: Velocity-vector-based grouping for vehicles

As a reminder it follows the definition of the dot product.

$$\left.\begin{matrix} u = (u_1, u_2) \\ v = (v_1, v_2) \end{matrix}\right] v * u = v_1 u_1 + v_2 u_2 \quad \text{Eq. 2}$$

The dot product can also be used to calculate the angle between 2 vectors:

$$\cos\theta = \frac{v * u}{\|v\|\|u\|} \quad \text{Eq. 3}$$

4.2) "Mobility-Based Clustering in VANETs using Affinity Propagation" by Christine Shea, Behnam Hassanabadi and Shahrokh Valaee

4.2.1) Introduction

The proposed algorithm, which is named APROVE (Affinity PROpagation for Vehicular networks), is a distributed mobility-based clustering algorithm that primarily focuses on cluster stability and utilizes the Affinity Propagation in a distributed manner. When talking about stability, we mean that within clusters we have:

- long clusterhead duration with low rate of cluster head change and
- long cluster member duration.

This will be achieved by forming clusters where clusterheads and their corresponding members will have both the minimum distance and the minimum relative velocity between them.

In this algorithm the position information on the vehicles is provided by the use of GPS.

4.2.2) Affinity Propagation algorithm

Affinity Propagation is a clustering method for data such as K-means or Euclidean Distance, but it does produce revolutionary faster clusters than the previously mentioned methods and with less errors than

the traditional techniques. More specifically, we can view our clustering problem as an attempt to minimize the distance between a certain data point and its assigned exemplar.

In Affinity Propagation, data points exchange messages with one another where they describe the current affinity that one data point has for choosing another data point as its exemplar. The algorithm receives as an input a function of similarities $s(i,j)$.

$s(i,j)$: indicates how well suited the data point j is to be the exemplar of data point i.

As it is conceivable in Affinity Propagation, we need to maximize the similarity $s(i,j)$ for every data point i and the chosen exemplar j. Each node i is also characterized by a self-similarity $s(i,i)$, a factor necessary for determining the number of exemplars that are identified. An individual node that is initialized with a larger self-similarity is more likely to become an exemplar and if all the nodes have the same initial self-similarity, they are equally likely to become exemplars. If we increase the common self-similarity input, we will produce more clusters.

We can have either responsibility $r(i,j)$ or availability $a(i,j)$ messages exchanged between the nodes.

- $r(i,j)$: is sent from node i to the candidate exemplar j and indicates how well suited is j to be the i's exemplar by taking into account competing potential exemplars.
- $a(i,j)$: is sent from candidate exemplar j back to node i. It indicates j's desire to be an exemplar for i by the data gathered.

The self-responsibility $r(i,i)$ and self-availability $a(i,i)$ are accumulated evidence that i is an exemplar. The update formulas for responsibility and availability are:

$$r(i,j) \leftarrow s(i,j) - \max_{j' \, s.t. j' \neq j} \{a(i,j') + s(i,j')\} \qquad \text{Eq. 4}$$

$$a(i,j) \underset{i \neq j}{\leftarrow} min\left\{0, r(j,j) + \sum_{\forall i' \notin \{i,j\}} max\{0, r(i',j)\}\right\} \qquad \text{Eq. 5}$$

$$a(j,j) \leftarrow \sum_{i' s.t. i' \neq j} \max\{0, r(i',j)\} \qquad \text{Eq. 6}$$

Their message updates have to be damped in order to avoid numerical oscillations that will prevent the algorithm's convergence. So we may use the following formula:

$$m_{new} = \lambda \, m_{old} + (1-\lambda) m_{new} \qquad \text{Eq. 7}$$

where $\lambda \in [0,1]$. Clustering is complete when the messages converge. When $r(i,i) + a(i,i) > 0$, the node become is able to determine about its clusterhead. Upon convergence, each node i's clusterhead is:

$$CH_i = \arg\max_j \{a(i,j) + r(i,j)\} \qquad \text{Eq. 8}$$

4.2.3) Proposed VANET clustering scheme, APROVE

In the proposed algorithm, named APROVE (Affinity PROpagation for Vehicular networks), it is used the basic idea from affinity propagation algorithm but in a distributed way. Each node transmits the responsibility and availability messages to its neighbors and then makes a decision on clustering independently. The result is that the decisions are taken in a distributed manner and simultaneously every node is only clustering with those in its one-hop neighborhood.

The algorithm, in order to function properly with the new position information, requires a new similarity function, which is defined as follows:

$$s(i,j) = -\left(\|x_i - x_j\| + \|x'_i - x'_j\|\right) \quad \text{Eq. 9}$$

$$x_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{Eq. 10}$$

$$x'_i = \begin{bmatrix} x_i + v_{x,i}\tau_f \\ y_i + v_{y,i}\tau_f \end{bmatrix} \quad \text{Eq. 11}$$

where:

- x_i is a vector of node i's current position,
- x'_i is a vector of node i's predicted future position.

The corresponding function predicts each node i's future position in τ_f (time future parameter) seconds from now, based on node i's current velocity $v_{x,i}$ in the x direction and velocity $v_{y,i}$ in the y direction. The τ_f parameter can be tuned for different types of mobility.

The self-similarities values can be initialized to the same preference value. This value must be set in such a way that it will minimize the number of clusters produced. It is possible, though, to assign a higher preference to certain vehicles (e.g. large trucks), which will make them more likely to become clusterheads.

A) Message Passing and the Neighbor List

Every node i will maintain a neighbor list $\boldsymbol{N_i}$, which has a neighbor list entry $\boldsymbol{N_i^j}$, for every neighbor j.

$$\boldsymbol{N_i^j} = \begin{cases} (x,y)_j : position\ vector\ of\ node\ j \\ (v_x, v_y)_j : velocity\ vector\ of\ node\ j \\ s(i,j): similarity\ for\ i\ and\ j \\ a(i,j) : last\ availabilty\ received\ from\ j \\ a(j,i) : last\ availability\ transmitted\ to\ j \\ r(j,i): last\ responsibility\ received\ from\ j \\ r(i,j): last\ responsibility\ transmitted\ to\ j \\ CH_{cnvg,j} : clusterhead\ converge\ flag\ for\ node\ j \\ CH_j : index\ of\ j's\ current\ clusterhead \\ t_{expire}: time\ that\ node\ j\ expires \end{cases} \quad \text{Eq. 12}$$

Each node j will periodically broadcast a **HELLO** beacon, which will contain its ID, position, velocity and current clusterhead. The hello broadcast period is defined as T_H. When node i receives a **HELLO** beacon message from node j, it calculates its current similarity with j, $s(i,j)$ (Eq. 9) and then updates its neighbor list with the new information. Messages are only considered by neighbors who are moving towards the same direction and the rest of them are discarded. The basic procedure is outlined in Procedure 1:

Procedure 1: Broadcast and Reception of HELLO Beacons

1. Every T_H, each node j broadcasts **HELLO** beacon: $\langle j, (x,y)_j, (v_x, v_y)_j, CH_j \rangle$
2. Each receiving neighbor, i, checks if j is travelling in the same direction
3. If true, i calculates similarity with j, $s(i,j)$
4. Node i adds/updates j's neighbor list entry, $\boldsymbol{N_i^j}$'s: $\langle j, (x,y)_j, (v_x, v_y)_j, s(i,j), t_{expire}, CH_j \rangle$

The broadcast period for availability and responsibility messages is defined as T_M. Every node i will calculate its responsibility with each neighbor j using Eq. 4. The value is damped with the previous transmitted responsibility (with $\lambda = 0.5$) and stored as $r(i,j)$ for each neighbor j in the responsibility array, $\boldsymbol{R_i}$, and broadcasts the array in the **RESP** packet. Each node i will calculate the availability with each neighbor j using

the update Eq. 5. Node i will store j's damped availability in $a(j,i)$ and accumulates all $a(j,i)$'s in the availability array, $\boldsymbol{A_i}$. This array is broadcasted in the **AVAIL** packet.

The **AVAIL** packet also includes the flag CH_{cnvg}. A node will become clusterhead when $r(i,i) + a(i,i) > 0$. For every iteration, each node i checks for this condition, and then sets the CH_{cnvg} flag accordingly. According to this flag, i may be considered or not by its neighbors as a potential clusterhead. The responsibility and availability broadcast procedure is outlined in Procedure 2:

Procedure 2: Broadcast of RESP and AVAIL messages

Every T_M, each node i will:

1. Calculate responsibility, $r(i,j)$ for each neighbor j
2. Update the damping factor: $r(i,j) = (1-\lambda)r(i,j)^{new} + \lambda r(i,j)^{old}$
3. Store responsibilities, $r(i,j)$, in array: $\boldsymbol{R_i}$
4. Calculate availability, $a(j,i)$ for each neighbor j
5. Update with damping factor: $a(j,i) = (1-\lambda)a(j,i)^{new} + \lambda a(j,i)^{old}$
6. Store availabilities, $a(j,i)$, in array: $\boldsymbol{A_i}$
7. Determine if converged to CH status: if $r(i,i) + a(i,i) > 0$, then set CH_{cnvg}
8. Broadcast the **RESP** packet: $\langle \boldsymbol{R_i} \rangle$
9. Broadcast the **AVAIL** packet: $\langle \boldsymbol{A_i},\ CH_{cnvg} \rangle$

On receiving a **RESP** or **AVAIL** packet from j, node i will search for its id in the $\boldsymbol{R_j}$ or $\boldsymbol{A_j}$ array and if it founds its id, it will read off its specific responsibility or availability message. These messages are stored in the received message fields, $r(j,i)$ or $a(i,j)$ of j's neighbor list entry, $\boldsymbol{N_i^j}$. If the received packet is of **AVAIL** type, node i will also update the $CH_{cnvg,j}$ field for j according to the CH_{cnvg} flag received. The procedure is outlined in the Procedure 3.

Procedure 3: Reception of RESP and AVAIL messages

Upon reception of a **RESP** or **AVAIL** packet from node j, node i will:

1. Search for its id, i in the $\boldsymbol{R_j}$ or $\boldsymbol{A_j}$ array
2. If a message addressed to i is found, update the $r(j,i)$ or $a(i,j)$ field in the neighbor list entry $\boldsymbol{N_i^j}$
3. Check if CH_{cnvg} flag is set, and update $CH_{cnvg,j}$ field in j's neighbor list entry $\boldsymbol{N_i^j}$

B) Cluster Formation and Maintenance

Clustering decisions are made periodically every CI (Clustering Interval). The T_M message period must be small enough to allow the algorithm to converge within the CI period. (Simulations have shown that a neighborhood of 40 nodes can converge in under 10 iterations. So, a T_M of 1s requires a minimum CI of 10s.)

Every CI, node i decides about its clusterhead by the use of Eq. 8. Node i only considers its neighbors with the $CH_{cnvg,j}$ flag set, which confirms that node j will become a clusterhead. If none of the neighbors have set their $CH_{cnvg,j}$ flag, node i will become its own clusterhead.

Between the clustering iterations of CI, it is performed cluster maintenance with period T_{CM} (cluster maintenance period). Every T_{CM}, node i clears its neighbor list from all the old entries by checking the t_{expire} fields and then checks whether its clusterhead is still in its neighbor list. If the CH has been lost, node i searches through its neighbors for current CHs to join, by finding neighbors with $CH_j = j$. The $CH_{cnvg,j}$ flag is not used in this case because it indicates the potential clusterheads for the next round. If there are found more than one CHs in the neighbor list, node i uses Eq. 8 to select the best one. If it can't find another neighbor that is currently a CH, it becomes its own clusterhead.

As far as it concerns the passing of the responsibility and availability messages, they are not reset between clustering iterations. So the algorithm can gain some memory and the nodes have preference to previous clusterheads. This results in fewer less frequent cluster changes. Secondly, the provided scheme doesn't require synchronization and each node can run independently from one another. In this asynchronous case, the exchanged availability and responsibility messages will be at most one period old and the performance of the algorithm won' t be effected. But in case of a lossy channel, messages can be older than one period, which may cause degradation on the algorithm's performance.

4.2.4) Notes about the algorithm

As mentioned in the initial paper, the algorithm requires enough time in the start in order to converge. For example in a simulation that lasts at about 500s, it can be used only the last 200s in order to survey the stability of the produced clusters.

As it is obvious by the description of the algorithm, self-similarity must be predefined on the network. This parameter could affect the final results of the simulation. For our experiments, all the nodes were assigned the same self-similarity.

Moreover, it is mentioned in the Procedure 2 and on the step 7 that "A node will become clusterhead when $r(i,i) + a(i,i) > 0$". Here we have to mention that when this requirement is implemented, all the nodes become clusterheads and there are no cluster members. Additionally, if this requirement is not implemented, then the nodes are simply "forwarders". Every node will rebroadcast the message that he has received, but a node will listen a new message only when it comes from its clusterhead. Specifically, a node decides about its clusterhead, without the potential clusterhead to have declared itself as clusterhead. In this case, the common scenario was that node i was considered CH for node j and node j was considered CH for node i. Both scenarios were implemented, but for the comparisons it was chosen the second scenario. This is the reason why there are so many CHs declared in Chapter 5 (comparison of the algorithms).

4.2.5) Files produced and description

- **zzz_main.cc**: As described in the section 3.4) Initial files and description. There is a call of the aproveCall function at the end of the file.
- **movement.h** & **movement.cc**: As described in the section 3.4) Initial files and description.
- **aproveCall.h** & **aproveCall.cc**: These files implement the basic algorithm that is described in the above paper on the Procedure 1, Procedure 2 and Procedure 3. **aproveCall.h** also contains the defined parameters mentioned on the above Procedures such as the damping factor λ.
- **neighborListEntry.h** & **neighborListEntry.cc**: These files contain the list of neighbors that a node retains as it was analyzed in Eq. 12.
- **m_helloBeacon.h** & **m_helloBeacon.cc**: These files simulate the information that is transmitted from one node to another when a Hello_Beacon message is transmitted form one node to its neighbors in order to inform them for his existence.
- **m_resp.h** & **m_resp.cc**: These files simulate the information that is transmitted from one node to another when a responsibility message is transmitted.
- **m_avail.h** & **m_avail.cc**: These files simulate the information that is transmitted from one node to another when an availability message is transmitted.

Figure 7 summarizes the files that were used in order to implement the aproveCall algorithm and the way the files were included from each other.

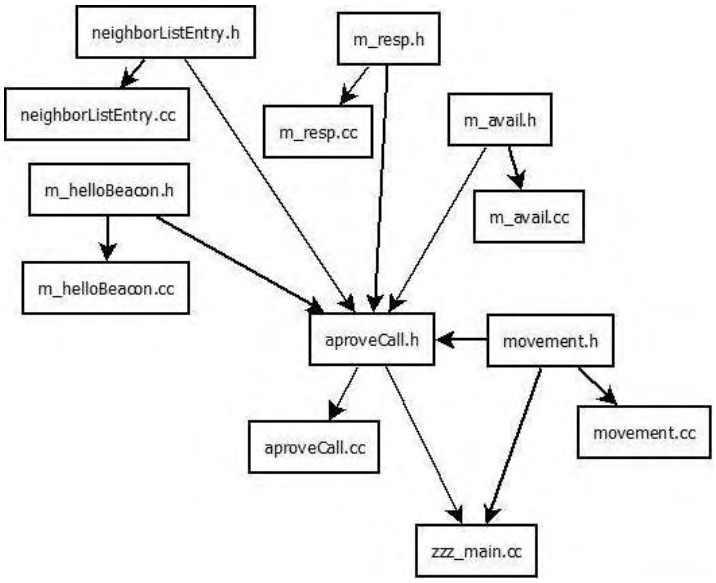

Figure 7: Files that were implemented in order to simulate the aproveCall algorithm and how they were included.

4.3) "A new Aggregate Local Mobility (ALM) Clustering Algorithm for VANETs" by Evandro Souza, Ioannis Nikolaidis and Pawel Giburzynski

4.3.1) Introduction

The key component of the algorithm is a variant of Aggregate Local Mobility (ALM) as the criterion triggering cluster re-organization and it is a beacon-based clustering algorithm whose objective is to extend the lifetime of a cluster. It has also incorporated a contention-based scheme to prevent over-eager re-organization of clusters when two clusterheads accidentally get in each other's range for a short period of time.

The algorithm is based on the observation that the probability that a node will be CH for a short period of time is higher than when the period is long. An increase in traffic density was observed to amplify the effect. It is clear that with the increase in the traffic density, more CHs will be in range of other CHs, and many of them will be forced to change their state. However, there is no compelling reason to force such a CH to change its state immediately: neighboring clusterheads may operate together for a certain amount of time.

The proposed algorithm brings a significant increase in the cluster's lifetime, thus improving network stability and in particular, decreasing the number of situations when nodes have to change their status within the cluster.

In this algorithm, we have the following states of a node:

1. Clusterhead (CH)
2. Cluster Member Node (MN)
3. Undecided Node (UN), for a node that hasn't decided yet if it should be a CH or a MN.

4.3.2) Aggregate Local Mobility algorithm

In this algorithm it is proposed the use of a measure of mobility to stabilize the CH state. However, the clusterheads may still change their status rapidly (and unnecessarily) if they get into one another's range.

The general concept of the aggregate mobility has been studied as a way to improve the stability of clusters. One scheme proposed uses the **received signal strength (RSS)** by the receiving node as an indication of the distance between the sender and the receiver. The ratio between two successive takes of that measure for the periodic hello messages provides an indicator of the relative mobility between the two nodes. Additionally, it was proposed to damper the trigger e.g. to delay re-clustering for a certain predefined amount of time when two clusterheads move into one another's range. By this way, accidental contacts between (otherwise stable) clusterheads will not cause unnecessary and intermittent re-organizations.

This algorithm shares the same general idea, but it employs a complicated set of rules.

1. We require that the clusterheads exchange more than one packet within a certain amount of time (known as the contention time) in order to begin considering a re-organization.
2. The node's decision regarding its status change is based on its perception of the aggregate local mobility (ALM). The CH with the lower ALM maintains its state, whereas the other changes it.
3. We also prevent a regular member (MN node) from immediately changing its status to CH when it stops receiving beacons from its last clusterhead and there is no other clusterhead in the neighborhood. When this scenario happens, the MN node will firstly move on to the UN state. This postpones the formation of a new clusterhead, which could trigger unnecessary re-clustering and also gives time to the MN node to detect another clusterhead that it could subscribe too.

By applying the previously mentioned, we conclude to the finite state machine of Figure 8. A node starts in the UN state where it listens to the wireless channel and periodically sends hello packets to announce its presence in the neighborhood. It also sets up a timer to wait for clusterheads to materialize in its range. Upon reception of a hello message from a clusterhead (HELLO_CH), the node immediately changes its state to MN. If no such message is received, the node will claim itself as a CH. While in MN, the node only changes its state to UN again in two circumstances:

1. no CH hello message has been received within the timeout interval (meaning that there is no clusterhead in the neighborhood),
2. there are no neighbors, which condition is detected via another timeout – while expecting any hello message.

The case of a node in CH state is more complex.

- If a clusterhead does not receive packets from any other clusterhead, it continues in the CH state.
- If the node detects that it has no neighbors (general hello timeout), then it will return to the UN state.
- The important part happens when two clusterheads get into one another's range and exchange packets. After receiving a packet from the other CH, the first clusterhead enters the connection mode (CCI). This means that it will wait for a certain amount of time (the connection time) to see if it receives another packet from the same CH.
    a) If that does not happen before the contention timer expires, the first message is simply ignored and the node continues as a clusterhead.
    b) Otherwise (there is a second HELLO_CH), the node has to decide whether to continue as a clusterhead or to change its state. The decision is based on the ALM weight (announced in the hello

messages). The node with the lower ALM will remain in the CH state (ALM_INF) , while the other will change to MN (ALM_SUP).

The ALM weight is calculated using GPS (or other similar technology) in order to assume that nodes know their position (within some easily tolerable error), instead of RSS, which is highly unreliable. The ratio between two successive takes of the distance between a node and its neighbor defines the relative mobility between the two. Specifically, the relative mobility of node Y with respect to node X is:

$$M_Y^{rel}(X) = \log\frac{Dist_{current}}{Dist_{previous}} = \log\frac{\sqrt{(x_X - x_Y)^2 + (y_X - y_Y)^2}}{\sqrt{(x'_X - x'_Y)^2 + (y'_X - y'_Y)^2}} \quad \text{Eq. 13}$$

where: $(x_X,\ y_X), (x'_X,\ y'_X)$ are the cartesian coordinates current and previous repsectively of node X and $(x_Y,\ y_Y), (x'_Y,\ y'_Y)$ are the cartesian coordinates current and previous repsectively of node Y.
The calculation of a node's ALM is the variance of the relative mobility over all neighbors, X_j, of a node Y:

$$M_Y = var_0\left(M_Y^{rel}(X_j)\right) = var_0\left(M_Y^{rel}(X_1), M_Y^{rel}(X_2), \dots, M_Y^{rel}(X_n)\right) = E[(M_Y^{rel})^2] = \frac{1}{n}\sum_{i=1}^{n}\left(M_{Y,i}^{rel}\right)^2 \quad \text{Eq. 14}$$

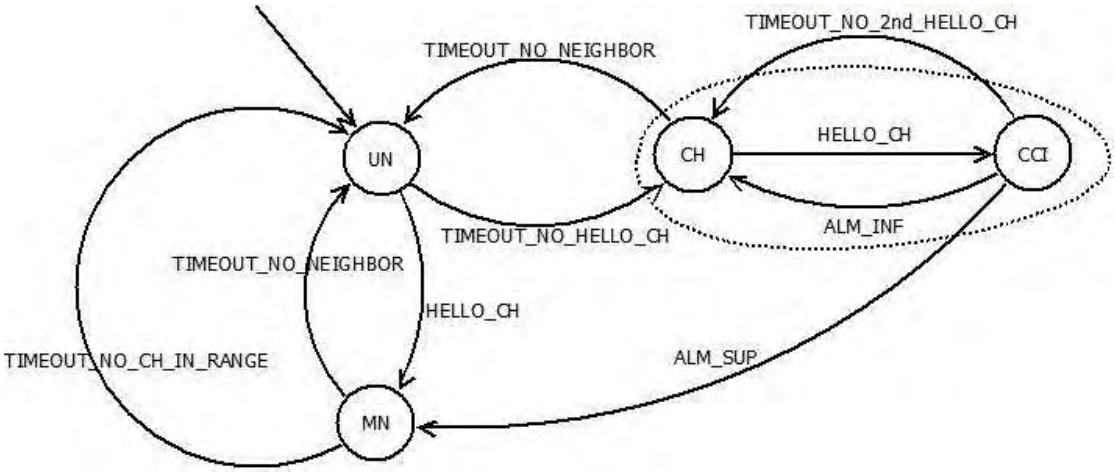

Figure 8: The proposed protocol for ALM algorithm. The transitions to the same state are not represented.

Two nodes moving closer together result in a negative $M_Y^{rel}(X)$, whereas, $M_Y^{rel}(X)$ is positive when they are moving away from each other. A lower variance in Eq. 14 suggests less mobility of the node in relation to its neighbors. The intuition behind the proposed scheme is that a node with less variance relative to its surroundings is a better (more stable) choice for CH.

4.3.3) Tools and Methodology

The algorithm requires to take into consideration and set the following parameters for each state (UN, CH, MN):

- Periodic hello-beacons interval, which probably will change its state.
- Periodic clean-ups interval, which represent the timeout where a node needs to take a decision in order to change its state.

The hello messages included the information necessary to calculate ALM: node ID, node status, node ALM and position (current and previous).

4.3.4) Notes about the algorithm and Implementation details

For the implantation of the algorithm, there were defined the following parameters:

- *TIMEOUT_NO_NEIGHBOR*
- *TIMEOUT_NO_HELLO_CH*
- *TIMEOUT_NO_SECOND_HELLO_CH*
- *TIMEOUT_NO_CH_IN_RANGE*
- *HELLO_MESSAGE*, as the period for the broadcast of the hello-beacon messages from the nodes.

The algorithm doesn't describe the kind of information that a node keeps and administrates. The node has to set internally timers for the above Timeouts. So each node has the following variables:

- *timer_neighbor*, which is used in CH & MN states,
- *timer_hello_ch*, which is used in UN state,
- *timer_second_hello_ch*, which is used in CCI state,
- *timer_ch_in_range*, which is used in MN state.

It also keeps its current CH and its current state, along with the alm value that calculates by the use of Eq. 13 & Eq. 14. Finally, the node is required to keep a vector of the hello-beacon messages that the node received recently (the current round / period of time).

We have to note that when a node searches for its CH, the states CH and CCI are a unified state and are recognized as CH state. Only when the node loses its property of being CH, will its MN node become a UN node to start searching for a new CH.

The ALM value is useful only if the node is a CH or a CCI node. In any other case, this field is not actually useful for the message and could be ignored.

Another thing that is not specified in the algorithm is what happens when a node receives more than one HELLO_CH messages from 2 or more different CHs. In such cases, he could choose in random.

Finally, it was chosen to add both the current and previous position of a node in the hello_beacon message. This could facilitate the implementation of the code and could also serve in avoiding scenarios where a new node emerges in a CH and the node doesn't have previous information to calculate the alm factor.

On the following table, it is presented a basic outline of the cases that were implemented in order to achieve the functionality of Figure 8.

```
for(i=0; i<numberOfNodes; i++){

        /*** UN_status ***/
        if(nodes[i].getStatus() == UNstatus){
                //If the node has received a HELLO_CH message → the node becomes MN to the CH.
                foundClusterhead = nodes[i].searchForClusterhead();
                if(foundClusterhead != -1){
                        //go to MN
                }
                //(Else) If there is a TIMEOUT_NO_HELLO_CH → the node becomes CH (of course, if it has neighbors)
                else if(nodes[i].getTimerHelloCH() <= 0){
                        if(nodes[i].sizeNeighborsVector() != 0){
                                //go to UN
                        }
```

```
                        else{
                                //stay in UN and update timers
                        }
                }
        }

        /*** MN_status ***/
        else if(nodes[i].getStatus() == MNstatus){

                //If no neighbors are detected and has passed the TIMEOUT_NO_NEIGHBOR→
                //the node becomes a UN.
                if(nodes[i].getTimerNeighbor() <= 0 && nodes[i].sizeNeighborsVector() == 0){
                        //go to UN
                }

                //there are neighbors detected
                else{
                        //If the old clusterhead is not still in range
                        foundClusterhead = nodes[i].searchForCertainClusterhead(nodes[i].getClusterhead());
                        if(foundClusterhead == -1){
                                foundClusterhead = nodes[i].searchForClusterhead();
                                if(foundClusterhead == -1){
                                        //If no CH in range is detected → the node becomes a UN.
                                        if(nodes[i].getTimerCHinRange() <= 0){
                                                //go to UN
                                        }
                                }
                                else{
                                        //stay in MN and update the timers
                                }
                        }

                        //Else if the clusterhead is still in range, simply update the parameters
                        else{
                                //stay in MN and update the timers
                        }
                }
        }

        /*** CH_status ***/
        else if(nodes[i].getStatus() == CHstatus){
                //If the node receives another Hello_Clusterhead message → it enters the CCI mode.
                foundClusterhead = nodes[i].searchForClusterhead();
                if(foundClusterhead != -1){
                        //go to CCI
                }

                else if(nodes[i].getTimerNeighbor() <= 0){
                        //If the size of the Neighbors_vector is 0 and the node has reached the
                        //TIMEOUT_NO_NEIGHBOR → then the node returns to UN.
                        if(nodes[i].sizeNeighborsVector() == 0){
                                //go to UN
                        }
                        else{
                                //stay in CH and update timers
                        }
                }
        }

        /*** CCI_status ***/
        else if(nodes[i].getStatus() == CCIstatus){

                //If the node doesn't receive a 2nd Hello_Clusterhead message &
                //it has passed TIMEOUT_NO_SECOND_HELLO_CH → it becomes CH again.
                foundClusterhead = nodes[i].searchForClusterhead();
                if(foundClusterhead == -1){
                        if(nodes[i].getTimerSecondHelloCH() <= 0){
                                //go to CH
                        }
                }
                //else, the node has received a second Hello_Clusterhead message and has to take some decisions.
```

```
                else{
                        float othersNodeAlm = nodes[i].returnAlmOfOtherCH(foundClusterhead);
                        //The node with the lower ALM will remain in the CH
                        if(nodes[i].getAlm() < othersNodeAlm){
                                //go to CH
                        }
                        //The node with the largest ALM will become MN
                        else{
                                //go to MN
                        }
                }
        }

        /*** ERROR ***/
        else{
                cout << "SOME UNEXPECTED ERROR OCCURED" << endl;
                exit(1);
        }

}
```

4.3.5) Files produced and description

- **zzz_main.cc**: As described in the section 3.4) Initial files and description. There is a call of the alm function at the end of the file.
- **movement.h** & **movement.cc**: As described in the section 3.4) Initial files and description.
- **alm.h** & **alm.cc**: These files implement the basic algorithm that is described in the above paper mostly on the Figure 8 and on 4.3.2) Aggregate Local Mobility algorithm and on Eq. 13, Eq. 14. **alm.h** also contains the defined parameters mentioned on the Figure 8.
- **node.h** & **node.cc**: These files contain the internal state of each node.
- **m_helloBeacon.h** & **m_helloBeacon.cc**: These files simulate the information that is transmitted from one node to another when a Hello_Beacon message is transmitted form one node to its neighbors in order to inform them for his existence.

Figure 9 summarizes the files that were used in order to implement the ALM algorithm and the way the files were included from each other.

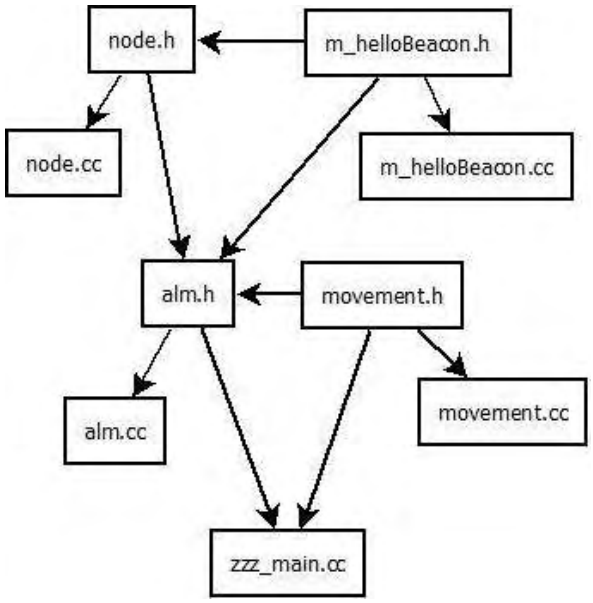

Figure 9: Files that were implemented in order to simulate the ALM algorithm and how they were included.

4.4) "Stable Clustering and Communications in Pseudolinear Highly Mobile Ad Hoc Networks" by Ehssan Sakhaee and Abbas Jamalipour

4.4.1) Introduction

To begin with, the following algorithm is designated for MANETs, instead of VANETs, although it could be also fit the scenarios of VANETs, but it fits mostly on the highway topology. On the paper, there are presented 2 algorithms who share common features. They are distinguished by the fact that the first algorithm knows the exact position of a node by the use of GPS, whereas the second node simply speculates the node's position by the use of the Doppler effect. Here, it is presented only the first algorithm, where we assume that we know the exact position of a node. The algorithm involves dynamic clusterhead election and incorporates cluster maintenance ta cater changes in network topology as time progresses. It is called **dynamic link duration clustering (DLDC)** and is based on the estimated **link expiration time (LET)**, which more accurately estimates the link duration based on the position and velocity parameters of nodes during the initial clustering phase.

Stability is defined in terms of cluster membership rather than changes to the clusterhead, such as clusterhead reelection and "ripple effects" due to reclustering. The proposed algorithm considers 2 phases:

- **Cluster formation** (initial clustering), in order to identify the most suitable nodes to take the clusterhead roles and in order to set up the initial memberships. Some nodes may not readily join any cluster after the initial clustering formation and are classified as "orphan" nodes.
- **Cluster maintenance** (progressive), which follows the initial cluster formation. It refers mostly to the nodes that may leave their cluster and clusterheads may come into range or may be destroyed.

The main types of nodes are:

- **clusterhead**, who is in charge of its corresponding cluster and formation (acceptance of nodes as members). A clusterhead has the knowledge of all its member-nodes inside the cluster.
- **cluster-member**, who are simply nodes that join the clusterhead. A cluster-member that belongs to more than one cluster is a **gateway** node, which is used for communication between clusters.
- **orphan**, who is an ex-member of a cluster that currently does not belong to any cluster.
- **NULL nodes**, which mostly characterizes the state of nodes before the cluster formation. The initial clustering formation only initiates between NULL nodes. Cluster maintenance occurs when nodes are not in the NULL state.

4.4.2) DLDC Algorithm

The cost metric used in DLDC is the **inverse link duration** (IDL), i.e. the inverse of Let ($\frac{1}{LET}$) (Eq. 15), which reflects a better (smaller) cost for more durable links and higher cost for less-durable links. The Hello packets in DLDC contain the position coordinates of the broadcasting node along with the state of the algorithm. Procedure 4 summarizes the steps that are followed for the initial phase of clustering.

Procedure 4: DLDC cluster formation

0. Each node is in the NULL state.
1. Each node broadcasts a *Hello* message, in the form *Hello(nodeID, NULL, position, velocity)* to its (1-hop) neighbors. nodeID → identifier (ID) of the broadcasting node.
2. Nodes in the NULL state that receive *Hello* messages from their neighbors will calculate the SUM of the ILD values (ILDS) of the Hello beacons. Then, they broadcast this to their 1-hop (NULL) neighbors in an *ILDS(nodeID, ILDS)* message.

3. Nodes that receive the ILDS messages compare them with their own ILDS.
   If their own is the smallest, they will broadcast *ClusterheadClaim(clusterID)* to their 1-hop neighbors.
   They will also change their status to CH.
   ClusterID → nodeID of the clusterhead claiming node.
4. Upon reception of *ClusterheadClaim* packet, a node may choose to join the cluster.
   - Specifically, a node can choose to become member of all clusters it receives a *ClusterheadClaim* packet from, and this node will become a gateway to all these clusters.
   - Otherwise, it may choose to compare the ILDs of all the *ClusterheadClaim* packets and only join the best one. (In this case, there will be no gateways.) (Non-overlapping initial cluster formation)
   - A third choice is to join the top m clusters with the smallest ILDs form which it received a *CluserheadClaim* packet.

   All nodes have to wait a predefined period *t* to collect the *ClusterheadClaim* messages before making a decision of becoming members. (If m=1, then t can be set to 0.)
5. Once a node wishes to join a cluster, it will send a *JoinRequest (nodeID, clusterID)*.
6. The clusterhead will send a *JoinAccept(nodeID, clusterID)*.
   nodeID → the requesting node ID
   clusterID → ID of the clusterhead
7. The node joins the cluster and periodically broadcasts a *ClusterMember(nodeID, clusterID)*, notifying its state to the neighborhood, and its presence to its associated cluster.

$$LET = \frac{-(ab+cd) + \sqrt{(a^2+c^2)r^2 - (ad-bc)^2}}{a^2+c^2} \qquad \text{Eq. 15}$$

where:

$$a = v_i \cos\theta_i - v_j \cos\theta_j = v_{i,x} - v_{j,x}$$
$$b = x_i - x_j$$
$$c = v_i \sin\theta_i - v_j \sin\theta_j = v_{i,y} - v_{j,y}$$
$$d = y_i - y_j$$

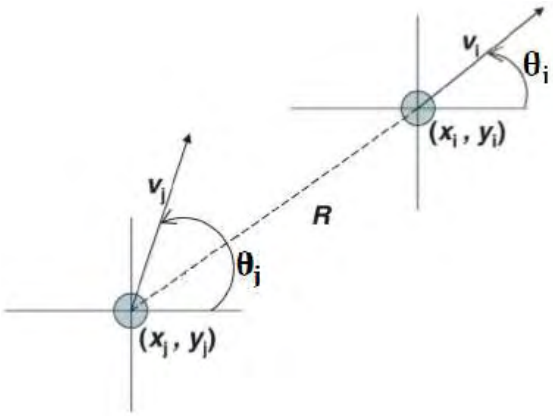

Figure 10: Parameters used in calculating the LET.

Additional stability is also possible in DLDC by setting a criterion for minimum link duration between pairs of nodes for clustering formation. In such a case, the LETs of pairs of nodes are first calculated upon reception of Hello messages and are only used for calculating the ILDS if the pairs of nodes will be within range for a minimum specified time. Hence, they are ignored and not used for a common cluster formation if their potential link duration is below this required time. This modified version of DLDC is called **DLDC - Super Stable** (DLDC-SS) and can provide more stable clusters even though it may leave some nodes without a cluster. The application of this algorithm is suitable in highly dense networks. However, it may not be suitable in sparse networks nor in scenarios where clustering is essential for addressing purposes.

4.4.3) Cluster Maintenance

Once the initial cluster formation takes place, clustering maintenance must follow. A clusterhead periodically sends *ClusterheadClaim(CusterID)* messages to its neighboring (1-hop) nodes. The clustermembers of the current cluster also send periodic *Clustermember(nodeID, clusterID)* back to the clusterhead, where the nodeID is the identifier (ID) of the broadcasting node and clusterID is the list of clusters of which the node is a member. A member node that does not receive periodic broadcast from a clusterhead will dissociate itself from that cluster. Likewise, the clusterhead will remove the member form its list of members if it does not receive periodic clustermember broadcasts. If a node is not part of any cluster, it becomes an orphan node and begins broadcasting periodic *JoinRequest(nodeID)* to find a new cluster to join. A clusterhead sends a *JoinAccept(nodeID, clusterID)* if it can allow the requesting node (with ID or nodeID) to join its cluster. In DLDC-SS, the LET of the orphan node relative to the clusterhead is considered. If the LET satisfies the minimum required possible (connection) duration, then a *JointAccept* message is sent to the targeted orphan node; otherwise, it is not sent. Once the node receives the *JointAccept* message and joins a cluster, it will begin broadcasting *Clustermember(nodeID, clusterID)* messages to its 1-hop neighbors. Upon receiving this message, the corresponding clusterhead will add this node to its list of clustermembers. A node that receives several *JoinAccept* messages from different clusterheads may become a gateway between these clusters. As the node is broadcasting the *Clustermember(nodeID, clusterID)* messages, the clusterheads will also learn about the gateway nodes and the clusters to which they have access.

Although the proposed clustering algorithm ensures that no two clusterheads are within range of each other in the initial clustering formation, this does not prevent two clusterheads from coming into each other's vicinity, which may cause cluster contentions. If such a scenario arises and clusterheads come within range of each other, the one with the bigger ILDS will give up its clusterhead if the 2 clusterheads have a relative ILDS that is less than a predetermined constant *K*. A small ILDS indicates that the clusterheads will remain within each other's range for a considerable period of time, and hence, it is worth one of them giving up its clusterhead status. Otherwise, if ILDS is large, the clusterheads may simply be passing by and soon leaving each other's range, and hence, it is not worth either of them giving up their clusterhead status.

In our algorithm, we assume that there is no limit to the number of nodes per cluster as the clusterhead should have sufficient capability to handle 1-hop neighbors. However, the clusterhead should be able to dissociate high-cost members in case of high level of congestion and heavy traffic load in the local cluster. Additionally, it is believed that the maximum number of clusterheads is not an effective measure for forming clusters as there are more predominant factors, such as frequency of communication (e.g., how often data are being exchanged between nodes), amount of data being transferred, and other similar traffic-related issues. The less-favorable member is sent a *Deregister(clusterID, nodeID)* by the clusterhead, where clusterID is the ID of the cluster that is forcing the deregistration, and nodeID is the ID of the node that is being forced to deregister. The member is, hence, forced to dissociate from its associated cluster. If the ex-member is not a member of any other cluster, it becomes an orphan node, where it will proceed with broadcasting JoinRequest packets and looking for new clusters to join. Deregistration may also occur to less-favorable gateway(s) to the same neighboring cluster(s).

4.4.4) In case a Clusterhead is destroyed or no Clusterhead in range

When a clusterhead is destroyed, its corresponding neighbors will no longer receive the periodic (Hello) messages and, hence, will dissociate from it. (In fact there are not broadcasted Hello messages but ClusterheadClaim and ClusterheadMember periodic messages.) If the nodes are not members of any other

cluster, they will become orphan nodes. They will then begin sending *JoinRequest* messages as a result. A mechanism is used to detect such a scenario where there are several orphan nodes within the same vicinity looking for clusters to join. If local orphan nodes receive too many *JointRequest* messages form their (orphan) neighbors, they conclude that there are no clusterheads nearby and may elect a new clusterhead for their vicinity. For example, when a node receives more than *k JoinRequests*, it will determine that a new clusterhead must be elected in the vicinity for which the clustering algorithm will need to reinitiate. Once this decision is made, the node that realizes the need for cluster reinitiation broadcasts a *Clusterinit(nodeID)* to its 1-hop neighbors. Neighbors that have not joined a cluster will set their states to NULL and begin the clustering-formation procedure using DLDC clustering algorithm (Procedure 4).

It follows the description of the Cluster Maintenance phase on Procedure 5:

Procedure 5: Cluster Maintenance

1. A clusterhead periodically sends *ClusterheadClaim(ClusterID)* to 1-hop neighbors.
   The clustermembers also periodically send *Clustermember(nodeID, clusterID)* back to clusterhead.
   - clusterID → the ID of the cluster that a node is member. When there are more than one clusterIDs, the node is considered to be a gateway.
2. If a member node does not receive periodic *ClusterheadClaim* message from a clusterhead, it will dissociate itself from that cluster.
   Similarly, if the clusterhead does not receive periodic *Clustermember* message from a certain node, it will remove the node from its list of members.
3. If a node is not part of any cluster, it becomes orphan. It then broadcasts periodic *JoinRequest(nodeID)* to find a new cluster.
   A clusterhead sends *JoinAccept(nodeID, clusterID)* to allow the requesting node to join the cluster.
4. When the orphan node receives the *JoinAccept*, it will start broadcasting *Clustermember(nodeID, clusterID)*.
   Upon reception, the clusterhead will also add the node to its list of clustermembers.
   - A node that receives several *JoinAccept* messages from different clusterheads will become a gateway.
   - (Clusterheads could get informed about the existence of gateways by the multiple clusterIDs on the *Clustermember* message.)
5. If 2 clusterheads come into each other vicinity,
   If they have a relative ILDS that is less than a predetermined constant *K*,
   Then, the clusterhead with the bigger ILDS will give up its clusterhead.
6. If no clusterhead is in range (no reception of *ClusterheadClaim* message) from multiple orphan nodes, they have to decide who is going to be the new clusterhead.
   If a local orphan node receives too many *JoinRequest* messages from its (orphan) neighbors, it will broadcast a *clusterInit(nodeID)* message to the 1-hop neighbors. Orphan nodes can change their state to NULL and can follow DLDC cluster formation (Procedure 4).

4.4.5) Notes about the algorithm

We have to mention here that in order to calculate ILDS when necessary, a node requires also to broadcast its position and its velocity in some messages. So, it is chosen to extend the ClusterMember message in order to include position and velocity information of a certain node. Specifically, the messages that were implemented (in different classes on C++) are:

- Hello(nodeID, Null, position, velocity)
- ILDS(nodeID, ILDS)
- ClusterheadClaim(clusterID)
- ClusterMember(nodeID, clusterID, position, velocity)
- JoinRequest(nodeID)
- JoinAccept(nodeID, clusterID)
- ClusterInit(nodeID)

Here, we have also to note that it was not implemented a Deregister. Such a message could be used in case that a Clusterhead has a lot of registered members.

Another issue that arouse during implementation, was related to the parameter ILDS and the calculation of the inverse LET. Specifically, there were many cases that we had 0/0 which is undefined and produces a "nan" result in the program. To better explain the problem, let's recall Eq. 15.

$$LET = \frac{-(ab+cd) + \sqrt{(a^2+c^2)r^2 - (ad-bc)^2}}{a^2+c^2}$$

where:

$$a = v_i \cos\theta_i - v_j \cos\theta_j = v_{i,x} - v_{j,x}$$
$$b = x_i - x_j$$
$$c = v_i \sin\theta_i - v_j \sin\theta_j = v_{i,y} - v_{j,y}$$
$$d = y_i - y_j$$

- if(a==0), then $LET^{-1} = \frac{c^2}{-cd+\sqrt{(cr)^2-(bc)^2}}$
- if(b==0), then $LET^{-1} = \frac{a^2+c^2}{-cd+\sqrt{(a^2+c^2)r^2-(ad)^2}}$
- if(c==0), then $LET^{-1} = \frac{a^2}{-ab+\sqrt{(ar)^2-(ad)^2}}$
- if(d==0), then $LET^{-1} = \frac{a^2+c^2}{-ab+\sqrt{(a^2+c^2)r^2-(bc)^2}}$
- if(a==0 && c==0), then $LET^{-1} = \frac{0}{0} = nan$

  This is the **only problematic situation**, but it occurs constantly in VANETs. To tackle with this problem, we use $LET^{-1} = 0$ in this case. The reason for this is that moving towards the same direction with the same velocities is considered to be beneficial for the stability of the cluster. By choosing as a CH the node with the smallest ILDS, this parameter should be set to 0.
- if(a==0 && b==0), then $LET^{-1} = \frac{c^2}{-cd+|cr|}$
- if(c==0 && d==0), then $LET^{-1} = \frac{a^2}{-ab+|ar|}$

Moreover, there was an issue with the precision in the representation of some numbers during the calculation of ILDS. Some values could become extremely small or large and could destroy the final output by producing inf (infinite) values. For this reason, it was chosen to use the following values:

```
if( abs(- (a*b + c*d) + sqrt( ( pow(a, 2) + pow(c, 2) ) * pow(r, 2)  - pow(a*d - b*c, 2) ) )< 1.3e-10){
        inverse_let = 100000000;
}
else{
        inverse_let = (pow(a, 2) + pow(c, 2)) / (- (a*b + c*d) + sqrt( ( pow(a, 2) + pow(c, 2) ) * pow(r, 2)  - pow(a*d - b*c, 2) ) );
}
```

We have also to mention that the period of time is large enough to handle the transmission of a message from a node and the reception of an answer or answers in order to form a cluster. In fact, every period of time ends up with the formation of clusters in such a way that the number of orphan or NULL nodes is the minimum possible.

Finally, this algorithm does not require some extra time in order to converge, whereas the 2 previously mentioned algorithms require some initial time in order to converge.

4.4.6) Files produced and description

- **zzz_main.cc**: As described in the section 3.4) Initial files and description. There is a call of the dldc function at the end of the file.

- **movement.h** & **movement.cc**: As described in the section 3.4) Initial files and description.
- **dldc.h** & **dldc.cc**: These files implement the basic algorithm that is described in the above paper mostly on Procedure 4 and Procedure 5. **dldc.h** also contains the defined parameters K and the ORHPAN_LIMIT mentioned on the Procedure 5.
- **node.h** & **node.cc**: These files contain the internal state of each node. They also contain defined the different states that a node can have.
- **m_Hello.h** & **m_Hello.cc**: These files simulate the information that is transmitted from one node to another when a Hello message is transmitted form one node to its neighbors in order to inform them for his existence. These messages are useful during the initialization of the process and when there are many orphan nodes at a certain neighbor, they need to follow cluster formation process in order to elect some clusterhead among them.
- **m_clusterheadClaim.h** & **m_ clusterheadClaim.cc**: These files simulate the information that is transmitted from a clusterhead node to inform its neighbors of its existence.
- **m_clusterMember.h** & **m_ clusterMember.cc**: These files simulate the information that is transmitted from a member node to its clusterhead. The clusterhead will listen to the message if he is still in range.
- **m_ilds.h** & **m_ilds.cc**: These files simulate the information that is transmitted from one node to another when ilds value is calculated and it is transmitted to the neighbors.
- **m_joinAccept.h** & **m_joinAccept.cc**: These files simulate the information that is transmitted from a clusterhead node to its member nodes in order to accept / confirm the node as a member node of the cluster.
- **m_joinRequest.h** & **m_joinRequest.cc**: These files simulate the information that is transmitted from a member node to its corresponding clusterhead in order to register itself to the cluster.
- **m_clusterInit.h** & **m_clusterInit.cc**: These files simulate the information that is transmitted from an orphan node to its neighbors when he has received more than ORHPAN_LIMIT m_joinRequest messages.

Figure 11 summarizes the files that were used in order to implement the ALM algorithm and the way the files were included from each other.

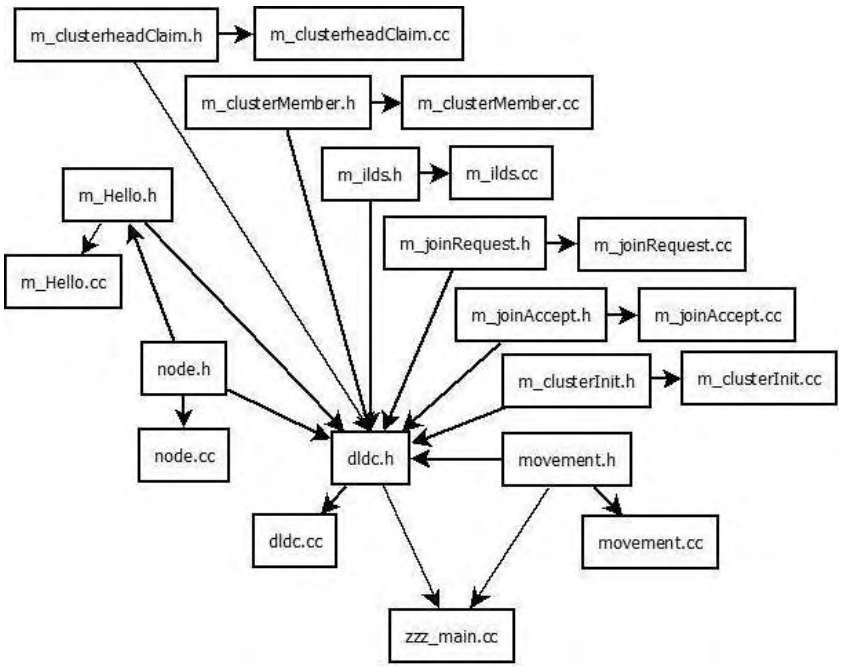

Figure 11: Files that were implemented in order to simulate the DLDC algorithm and how they were included.

Chapter 5 – Comparison of Algorithms for Stable Clustering in VANETs

This Chapter is dedicated to the comparison of the algorithms.

The following table summarizes some of the basic elements for the implementation of the algorithm.

Parameter for comparison	Algorithms		
	APROVE	ALM	DLDC
Difficulty in implementation	Very difficult (too many cases that had to be considered)	Normal, the simplest of the 3 We had to consider carefully the cases and the order that had to be checked.	Normal Problem with the calculation of LET Too many messages
Different types of broadcasted messages	3 Types: Hello Beacon Message, Availability message, Responsibility Message	1 Type: Hello Beacon message	7 Types: Hello Beacon message, ILDS message, ClusterheadClaim message, ClusterMember message, JoinRequest message, JoinAccept message, ClusterInit message.
Different types of nodes	Clusterhead and Clustermber but all the nodes must act as "forwarders"	3 types: Clusterhead (CH & CCI), CluserMember node, Undefined Node	4 types: Clusterhead, Clustermember and some of them are Gateways, Orphan Nodes, NULL nodes
Parameters that had to be considered	TAF_F 5 T_H 3 T_M 1 CI 10 T_CM 1 DAMPING_L 0.5 EXPIRE_TIME_DEFAULT 4 SELF_SIMILARITY 1.0 (defined in "aproveCall.h")	TIMEOUT_NO_NEIGHBOR 5 TIMEOUT_NO_HELLO_CH 5 TIMEOUT_NO_SECOND_HELLO_CH 5 TIMEOUT_NO_CH_IN_RANGE 5 HELLO_MESSAGE 1 (defined in "alm.h")	ILDS_K 10 ORPHAN_LIMIT 3 (defined in "dldc.h")

The following table summarizes the numeric results that were produced by the 2 experiments, as presented in 3.3) The Simulation Networks, that were simulated with the algorithms.

Parameter for comparison	Algorithms		
	APROVE	ALM	DLDC
Highway (simple-road) topology	**Number of nodes: 100 Time Periods:501 Transmission radius: 46.0936**		
Average number of messages per minute	7413.01	3175.12	224.052
Average number of clusters per minute	76.3174	6.182	4.99401
Average number of cluster size per minute	30.38	23.6823	19.6008
Average number of messages per 3 minutes	22239	9506	672
Average number of clusters per 3 minutes	228	18	14
Topology of a city	**Number of nodes: 100 Time Periods:501 Transmission radius: 62.7284**		
Average number of	5829.78	2602.32	283.92

messages per minute			
Average number of clusters per minute	76.8503	11.8808	13.8762
Average number of cluster size per minute	23.544	11.2261	3.83433
Average number of messages per 3 minutes	17489	7791	851
Average number of clusters per 3 minutes	230	35	41
Comments	A lot of messages that are broadcasted. Too many clusters.	It seems to provide good coverage of nodes in both scenarios of the city topology and the highway topology. There are enough broadcasted messages among the nodes but not too many.	In case of the city topology: (Average number of clusters per minute)x(Average number of cluster size per minute) < (Number of Nodes) ➔ which means that **not** all nodes are covered by some cluster. There are many nodes that are not covered by any cluster. However, in the highway topology, the algorithms provides pretty good clustering. In case of the highway, the number of broadcasted messages is smaller than those of the second algorithm. Conclusively, this algorithm is not preferable for a city topology (where also the number of messages that are broadcasted is much bigger).

References

1. Basics of ad hoc networks http://en.wikipedia.org/wiki/Wireless_ad-hoc_network
2. Basic on Clustering http://inf-server.inf.uth.gr/courses/CE522/findex_2010.htm, Dimitrios Katsaros
3. MANETs and VANETs http://en.wikipedia.org/wiki/Main_Page
4. Arzoo Dahiya, Dr.R.K.Chauhan "A Comparative study of MANET and VANET Environment", Journal of computing, Volume 2, Issue 7, July 2010, ISSN 2151 - 9617
5. Rituparna Ghosh, Stefano Basagni "Mitigating the impact of node mobility on ad hoc clustering", Wirel. Commun. Mob. Comput. 2008; 8:295–308
6. P. Basu, N. Khan, T.D.C. Little, "A Mobility Based Metric for Clustering in Mobile Ad Hoc Networks", MCL Technical Report No. 01-15-2001
7. Brad Karp, H. T. Kung "GPSR: Greedy Perimeter Stateless Routing for Wireless Networks", MobiCom 2000
8. Basic on SUMO: http://sumo.sourceforge.net/doc/current/docs/userdoc/Networks/SUMO_Road_Networks.html
9. SUMO tutorial http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Main_Page
10. Trace Exporter http://sourceforge.net/apps/mediawiki/sumo/index.php?title=Tools/TraceExporter
11. "Mobility-Based Clustering in VANETs using Affinity Propagation" by Christine Shea, Behnam Hassanabadi and Shahrokh Valaee
12. "A New Aggregate Local Mobility (ALM) Clustering Algorithm for VANETs" by Evandro Souza, Ioannis Nikolaidis and Pawel Giburzynski
13. "Stable Clustering and Communications in Pseudolinear Highly Mobile Ad Hoc Networks" by Ehssan Sakhaee and Abbas Jamalipour
14. "A Stable Routing Protocol to Support ITS Services in VANET Networks", by Tarik Taleb, Ehssan Sakhaee, Abbas Jamalipour, Kazuo Hashimoto, Nei Kato and Yoshiaki Nemoto.