UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

A KINETIC DISTANCE-TO-MEAN BASED ROUTING ALGORITHM FOR VEHICULAR AD-HOC NETWORKS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

ROMAIN SAULAS
Norman, Oklahoma
2017

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/52948

A KINETIC DISTANCE-TO-MEAN BASED ROUTING ALGORITHM FOR VEHICULAR AD-HOC NETWORKS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

_____
Dr. Mohammed Atiquzzaman, Chair

_____
Dr. Sridhar Radhakrishnan

_____
Dr. Changwook Kim

© Copyright by ROMAIN SAULAS 2017
All Rights Reserved.

Acknowledgements

I would like to express my sincere gratitude to the members of my Thesis committee for their advice on my thesis, and a special thank you to my advisor, Dr. Atiquzzaman, for all of his encouragement, guidance and instruction throughout the obtaining of my degree. Finally, I would like to thank Ms. Perez Woods for all her support.

Table of Contents

Acknowledgements iv

List of Tables ix

List of Figures x

Abstract xii

Chapter 1: Introduction 1

1.1 Introduction 1

1.2 VANET Principle and Applications 1

1.3 Hardware in VANET 3

1.4 WAVE IEEE 802.11p Standard 5

1.5 The routing problem in VANET 6

1.6 Problem Statement 7

1.7 Contribution 8

1.8 Thesis organization 8

Chapter 2: Literature Survey 10

2.1 Unicast, Multicast and Broadcast 10

2.2 Proactive, Reactive and Hybrid protocols 11

2.2.1 Proactive Protocols 11

2.2.2 Reactive Protocols 12

2.2.3 Hybrid Protocols 12

2.3 Categorization of next hop selection 13

2.3.1 Distance-based 13

2.3.2 Best Quality Link 14

2.3.3 Most Demanding node 14
2.3.4 Probability based forwarding 14
2.3.5 Backbone node 15
2.3.6 Stochastic method 15
2.3.7 Counter based 15
2.3.8 Distance to mean 16
2.4 Types of routing protocols 16
2.4.1 Topology Based Routing 17
2.4.2 Position Based Routing 23
2.4.3 Cluster Based Routing 25
2.4.4 Trajectory Based Routing 27
2.4.5 Geographic and Broadcast Routing 29
2.5 Conclusion 38
Chapter 3: VANET Kinetic Distance-To-Mean Algorithm (KDTM) 41
3.1 Distance-To-Mean Algorithm (DTM) 41
3.2 Kinetic Mobility Management Applied to Vehicular Ad Hoc Network. 45
3.2.1 Kinetic Node Degree 45
3.3 Beacon broadcasting rules 49
3.4 Proposed Kinetic Distance-To-Mean Protocol (KDTM) 51
3.5 Conclusion 53
Chapter 4: Distance-To-Mean (DTM) Implementation in NS3 54
4.1 Implementation packaging details 54
4.2 Protocols details and implementations 55

4.3 Conclusion 58
Chapter 5: Kinetic Distance-To-Mean (KDTM) Implementation in NS3 59
5.1 Implementation packaging details 59
5.2 Protocols details and implementations 60
5.3 Conclusion 63
Chapter 6: Simulation Scenario 64
6.1 Route topology and Mobility Scenarios 64
6.1.1 Urban scenario 64
6.1.2 Highway scenario 65
6.2 Network Simulation Implementation details 65
6.2.1 Node Creation 66
6.2.2 Devices Creation and WAVE 802.11p Protocols Simulation Model 66
6.3 Conclusion 67
Chapter 7: Results 68
7.1 Vehicular scenario simulation and simulation tools 68
7.2 Performance Parameters Evaluated 70
7.3 Results 71
7.3.1 Highway scenario 71
7.3.2 Urban scenario 76
7.4 Conclusion 80
Chapter 8: Conclusion and future work 82
Appendix A: SUMO urban map generation 90
Appendix B: DTM algorithm code 92

Appendix C: KDTM algorithm code 113

Appendix D: DTM ns3 main for urban and highway scenario 139

Appendix E: KDTM ns3 main for for urban and highway scenario 151

List of Tables

TABLE 1: SUMMARY OF ALL PROTOCOLS DESCRIBED IN THE LITERATURE SURVEY..... 39

TABLE 2: HIGHWAY SCENARIO PARAMETERS 71

TABLE 3: PROTOCOL SIMULATED IN HIGHWAY SCENARIO. 72

TABLE 4: URBAN SCENARIO PROTOCOL MEASURED. 76

TABLE 5: URBAN SCENARIO PARAMETERS. 77

List of Figures

FIGURE 1: VANET COMMUNICATION PATTERN. 2

FIGURE 2: ON BOARD UNIT. 3

FIGURE 3: ON-BOARD DIAGNOSTIC II READER. 3

FIGURE 4: ROAD-SIDE UNIT. 4

FIGURE 5: TYPES OF COMMUNICATION FOR EACH TYPE OF APPLICATION. 7

FIGURE 6: ILLUSTRATION OF DTM HEURISTIC FROM [1]. 42

FIGURE 7: DTM ALGORITHM. 44

FIGURE 8: STABILITY OF LINK IJ..... 46

FIGURE 9: DEGREE OF THE LINK IJ BETWEEN T_FROM AND T_TO. 47

FIGURE 10: NODE DEGREE AND KINETIC DEGREE. 49

FIGURE 11: CONSTANT DEGREE DETECTION. 50

FIGURE 12: IMPLICIT DETECTION. 50

FIGURE 13: ADAPTIVE COVERAGE DETECTION. 51

FIGURE 14: VARIATION OF THRESHOLD FUNCTION WITH THE PARAMETERS. 52

FIGURE 15: URBAN SCENARIO MAP (200 VEHICLES). 65

FIGURE 16: REACHABILITY IN HIGHWAY ENVIRONMENT. 73

FIGURE 17: NUMBER OF REBROADCAST PER COVERED NODE IN HIGHWAY ENVIRONMENT. 73

FIGURE 18: OVERHEAD FROM WARNING MESSAGES IN HIGHWAY ENVIRONMENT. 74

FIGURE 19: OVERHEAD FROM HELLO MESSAGES IN HIGHWAY ENVIRONMENT. 75

FIGURE 20: GLOBAL OVERHEAD..... 76

FIGURE 21: REACHABILITY IN URBAN ENVIRONMENT. 77

FIGURE 22: NUMBER OF REBROADCASTING PER COVERED NODE IN URBAN ENVIRONMENT. 78

FIGURE 23: OVERHEAD GENERATED BY WARNING MESSAGES IN URBAN ENVIRONMENT. 78

FIGURE 24: OVERHEAD GENERATED BY HELLO MESSAGES IN URBAN ENVIRONMENT. . 79

FIGURES 25: GLOBAL OVERHEAD IN URBAN ENVIRONMENT. 79

Abstract

Vehicular Ad-hoc Network (VANET) allows vehicles to send information to each other or to roadside equipment in an instant and a wireless manner. It represents a considerable step forward in terms of transportation. Indeed, many applications could benefit from instantly transmitting data such as video and music streaming. Internet connectivity could also be available to the passengers. The main objective and improvement aimed by VANET is a security matter. Warning messages for weather conditions, traffic accidents and fuel consumption instantly transmitted between vehicles would improve road safety. However, one of the critical issues in VANET is the dissemination of messages throughout the network. Indeed, routing algorithms are a key issue because of the high mobility and scalability of VANET compared to other ad-hoc networks.

This thesis contributes to the research area by firstly presenting a survey over most of the routing protocol used in VANET and classifying them according to the algorithm used. Secondly, this thesis develops more on the broadcasting algorithm category and particularly on the distance to mean heuristic method that presents good results in terms of reachability and bandwidth consumption. However, distance to mean protocol needs periodic beacons that creates a consequent overhead compared to instant rebroadcasting algorithm. Therefore, this thesis improves this method by reducing overhead using a bio inspired kinetic graph model proposed in the literature. The proposed modified algorithm has been implemented, simulated and evaluated in the network simulator NS3. Results show a lower overhead while preserving a good reachability.

Chapter 1: Introduction

1.1 Introduction

The arriving of Vehicular Network (VANET) in the transportation area is a great step forward. It allows vehicles to instantly send messages to other vehicles or infrastructures. One of the main objective of VANETs is the safety message dissemination which rely on broadcast communication, among vehicles. Therefore, the question of routing algorithm is an important one in VANETs and the literature on the topic provides a large variety of algorithm. Among those, a next rebroadcasting node selection technique named distance-to-mean (DTM) presents good results in terms of reachability and number of rebroadcast per nodes. However, it presents a higher overhead than most of instant rebroadcasting algorithm because of the periodic sending of beacon. Therefore, in this thesis we want to provide a solution using the Kinetic Graph Model algorithm to decrease the overhead of DTM.

In this chapter, we will present details of VANETs and the different protocols and hardware involved in it, the routing problem in VANETs and finally the statement of this thesis.

1.2 VANET Principle and Applications

Vehicular Ad-hoc Network (VANET), the arriving technology installing communications between vehicles on the roads, represents the future of transportation. Indeed, the transit of data between vehicles and infrastructures can provide infinite possibility of application impacting directly people's lives. Although comfort application (such as infotainment application) have a real potential using VANET, it is the area of safety applications that gathered the biggest amount of research. Surely, the decreasing

of number of accidents on the road is one the main objective of VANETs. To accomplish this goal, a VANET allows a car that detect a hazardous situation to warn other drivers on the road. For that to happen, a safety message would be generated and transmitted to other vehicles. With this type of communication, drivers can handle abnormal situation such as road or weather condition and crash, breakdown or emergency vehicle on site warning. VANET introduces three types of communications, vehicle to vehicle (V2V), vehicle to infrastructure (V2I), vehicle to pedestrian (V2P) and their respective inverse (I2V and P2V).

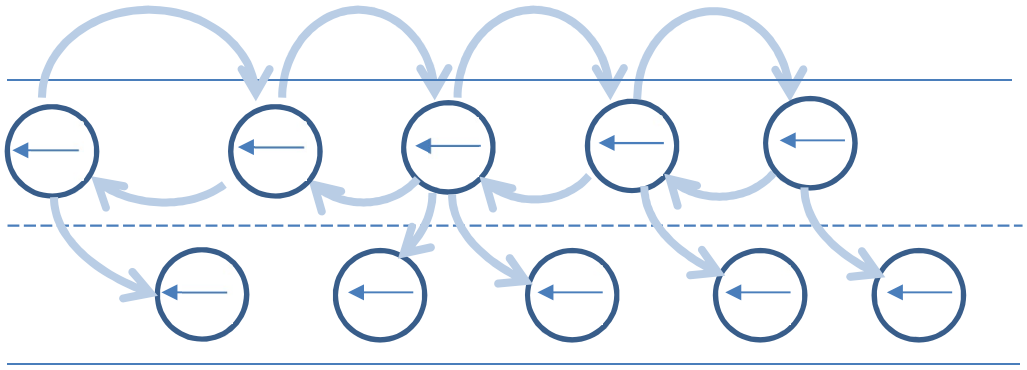


Figure 1: VANET Communication pattern.

The communication in VANETs requires information dissemination between all the cars on the road. The objective is to propagate information detected by a vehicle backward, for crash detection for example, and forward, for emergency vehicle for example. Figure 1 represents the types of propagation and the communication pattern in VANETs where each circle represents a car and the arrow indicates its direction of travel on the road. These two types of dissemination require rebroadcasting to transmit safety messages to all nodes and deal with several critical characteristics to be efficient. Indeed, safety

message dissemination is time sensitive, requires a high priority, a small payload and a high reachability.

VANET requires several hardware to function correctly. Those are presented in the next section.

1.3 Hardware in VANET



Figure 2: On Board Unit.

Because VANET main characteristic is to allow each vehicle to transmit information about itself and gather some from other vehicles, each vehicle must be equipped with specified hardware. Therefore, each vehicle transports an On-Board Unit (OBU) as shown in Fig. 2. OBUs are charge of handling all communication such as V2V, V2I and I2V. Those communication are effected using the Dedicated Short-Range Communication (DSRC) detailed in the next section of this paper.

The OBU is connected to an On-Board Diagnostic II Reader (OBD II Reader) as shown in Fig. 3, which is a chipset that stream vehicle information over Bluetooth. It transmit for example, the vehicle position, velocity, brake status and gear position. Information are then processed by the application running on the OBU and can be transmitted to other vehicles over DSRC.



Figure 3: On-Board Diagnostic II Reader.

Finally, VANET uses fixed nodes called Road Side Units (RSUs) as shown in Fig. 4. Those should be placed at certain strategic location so that they can be connected to a Traffic Control Center (TCC) over the Internet. RSUs can also communicate with all OBU over DSRC with V2I and I2V communication.



Figure 4: Road-Side Unit.

Now that we detailed the different hardware required to handle communications, we will explain in more detail the DSRC protocol and the communication standards used in VANETs.

1.4 WAVE IEEE 802.11p Standard

The Wireless Access for Vehicular Environments (WAVE) is the main protocol used to handle communication in VANET. WAVE is based on the WiFi IEEE 802.11 used in Mobile Ad-hoc Networks (MANET) but since VANET are characterized as highly scalable networks with a high mobility, new protocols were necessary. Therefore, WAVE defines several standards such as IEEE 802.11p, 1609.1, 1609.2 1609.3 and 1609.4.

- **IEEE 802.11p:** This standard is the Dedicated Short-Range Communication (DSRC) protocol. It covers the physical layer of the protocol stack. Derived from the IEEE 802.11 but with specifics characteristics, DSRC offers communication up to 1000 meters over the 5.9GHz band.
- **1609.1, Core Systems:** Defines recommendations for the application layer to use the WAVE protocol correctly.
- **1609.2, Security:** Defines the layer that handle security over communication and application in VANETs.
- **1609.3, Network Services:** Defines the layer that handle communication stacks and Transport and Network layers.
- **1609.4, Channel Management:** Defines the layer that handle multi-channel communications. Indeed, multi-channel is available with two types of channel:
    - **Control Channel (CCH):** Used for security matters, this channel offers low delay services and aims to transmit security messages to the network.
    - **Service Channel (SCH):** Used for services such as entertainment or non-safety dedicated communication. Six SCHs can exist in parallel

but each SCH needs the establishment of a communication between vehicles over CCH before being used.

WAVE defines the transmission protocol over DSRC and guidance on how to handle those communication. However, WAVE does not define the way a packet is disseminated from the sender node to the destination one over the network. It does not deal with the routing problem.

1.5 The routing problem in VANET

Since VANET presents different characteristics than MANET, such as highly scalable network with highly mobile nodes, MANET's routing protocol present poor reliability results in VANET. Principally in terms of network overhead and end-to-end delay which are critical objectives for a VANET. These criteria can be explained in more details as follow:

- **Low Bandwidth consumption (low overhead):** In order to adapt highly scalable network, the bandwidth consumption must be maintained at a minimum to be able to handle new communication introduced by nodes at any time and with a minimum number of collisions.
- **High reachability:** In case of warning messages, the whole network is concerned, then the number of node reached by a message over the number of concerned node is an important criterion.
- **Low end-to-end delay:** The time taken for a packet to reach destination is also an important matter especially with safety messages.

Accordingly, VANETs present more critical performance requirement than MANETs but the same main characteristic as every Ad-Hoc network. Which means, no

infrastructures dedicated to routing are present in the network. Therefore, every node deals with the routing. Then the routing problem could be defined as the way to handle message dissemination in highly mobile and scalable Ad-hoc networks, with respect of low bandwidth consumption, high reachability and low end-to-end delay.

The routing problem in VANETs being introduce, we will now define the problem statement.

1.6 Problem Statement

In the field of messages dissemination in VANETs, we can categorize algorithms as shown in Fig. 5, where the type of communication used is chosen in accordance with the type of application intended.

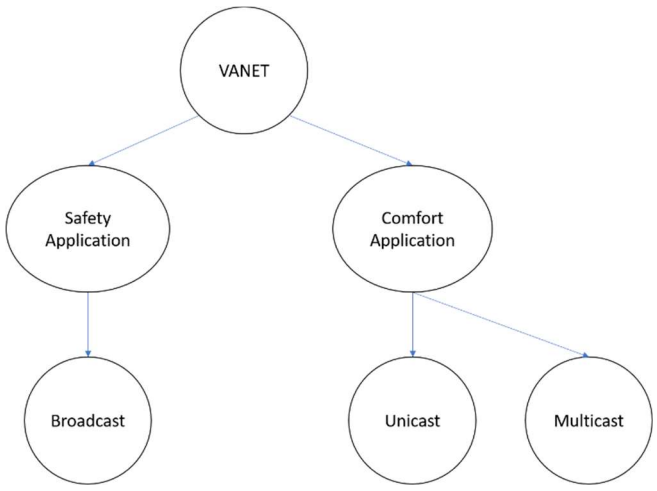


Figure 5: Types of communication for each type of application.

In this paper, we will focus on safety message dissemination, therefore, to broadcasting protocol. Broadcasting routing implies that every car is concerned by the message sent. Thus, for a message to be transmitted to every car, we need to select some of them to rebroadcast the received message. Consequently, we need a rebroadcasting

protocol that will decide which vehicle will forward the messages. This process is called the next-hop selection.

In this paper, we will make a complete literature survey on routing techniques and algorithms in VANETs for Broadcast, Unicast and Multi-cast communication. Then we will focus on one type of broadcasting method and study its bandwidth consumption and decrease it using a technique based on kinetic graph. Finally, we will use networks simulation tools to evaluate and validate the new algorithm presented.

1.7 Contributions

Contributions of this Master's Thesis are listed below:

- We propose an exhaustive survey of main algorithms designed in VANETs with a categorization under their types of routing and next-hop selection methods.
- We implement the Distance-To-Mean (DTM) Broadcasting algorithm in the Network Simulator software NS3 for Urban and highway scenarios.
- We propose a Kinetic Distance-To-Mean (KDTM) routing protocol to Reduce the bandwidth consumption of DTM algorithm, based on the Kinetic graph model.
- We implement the Kinetic Distance-To-Mean (KDTM) routing protocol in the Network Simulator software NS3 for urban and highway scenarios.
- We evaluate the KDTM algorithm on the simulated scenarios and compare the results to the DTM algorithm.

1.8 Thesis organization

To present the contributions, this thesis firstly presents a literature survey in Chapter 2 listing algorithms studied in VANET. Secondly, Chapter 3 introduce the KDTM algorithm proposed in this thesis. Based on the DTM algorithm and on the Kinetic graph

model. Thirdly, Chapter 4 presents the implementation of DTM in the Network Simulator NS3, and Chapter 5 the implementation of KDTM. Chapter 6 explains the definition of a simulation scenario in NS3. Chapter 7 presents the results, evaluation and validation of the KDTM algorithm. Finally, Chapter 8 presents the conclusion and the futures works.

Chapter 2: Literature Survey

A routing protocol defines the way the dissemination of a message in a network is handled. It defines the creation of a route from the source of the message its destination. In Ad-Hoc networks, every node is responsible to deal with the routing and do not rely on specified devices like network with infrastructure. This literature survey will firstly identify several ways to classify routing algorithms, secondly, it will present the routing protocols available in VANETs. Finally, a table will summarize this chapter by classifying the algorithms according to their different routing techniques. In the next section, to introduce routing algorithms, we will present the three different types of communication that routing algorithms can use: unicast, multicast and broadcast.

2.1 Unicast, Multicast and Broadcast

Those three types of communication define the number of point concerned by a transmitted message in a network. Unicast describes a communication between two points, the sender and the receiver. Multicast defines a communication between a sender and several receivers. Finally, broadcast describes a communication where the sender's message is sent to all the other nodes of the network. In VANETs, the three types of communication are used, depending on the types of application. Environment and entertainment application will use more often unicast and multicast because the message does not concern every vehicle. On the other Hand, safety application will mainly use broadcast communication to reach all nodes of the network.

Unicast and multicast transmissions need the establishment of a communication between the sender and the receiver(s) before the transmission. The absence of routing devices (such as gateway) in ad-hoc network implies that the communication will consist

of a succession of hops from the sender to the receiver(s). Where each node in the path will forward the message until it reaches its destination(s). The way the algorithm defines this path is called route discovery and is the first phase for most of unicast and multicast protocol in VANETs. Most of the time it uses broadcasting methods during this phase. Therefore, even if this thesis focusses more on safety application and on broadcasting algorithms, we will also go through some unicast and multicast protocols that use broadcasting techniques in their first phase. Finally, the results obtained for a new broadcasting algorithm could also impact the efficiency of unicast and multicast algorithm by improving the route discovery phase.

2.2 Proactive, Reactive and Hybrid protocols

Routing protocols can be classified under different criteria. From the way it handles routes in the network to the way it discovers those routes. In this section, we identify three behaviors defining when a route is established and maintained in the network. Those three types are proactive protocols, reactive protocols and hybrid protocols and will be detailed below.

2.2.1 Proactive Protocols

Proactive protocols use a route discovery phase before sending any data. Indeed, routes are calculated and maintained up to date continuously by transmitting periodic routing information on the network. Each time a node wants to transmit a packet, the packet is sent only if a route to the destination is established. Otherwise the packet will wait in queue until a route has been found. Those types of protocols are difficult to maintain in highly dynamic and scalable network such as VANETs. Therefore, they require a significant amount of routing information to be transmitted, increasing

significantly the bandwidth consumption. Accordingly, Proactive protocols are not the most suitable for VANETs. Examples of proactive protocols such as Destination Sequenced Demand Vector (DSDV) and Optimized Link State Routing Protocols (OLSR) are detailed later in this chapter.

2.2.2 Reactive Protocols

For reactive protocols, the route discovery phase is initiated only when a packet needs to be sent over network. If a route is found it will be maintained using maintenance route packets sent periodically until the destination is not reachable. Reactive protocols present lower overhead than proactive protocols, but the end to delay is more important due to the route discovery phase started every time a packet needs to be send. Therefore, reactive protocols present also some inconvenient in VANETs where the transmission require a low end to end delay. Examples of reactive protocols such as Ad-hoc OnDemand Distance Vector (AODV), Distance Vector Routing (DSR) are detailed later in this chapter.

2.2.3 Hybrid Protocols

Hybrid protocols are designed to compensate the overhead of proactive protocols And the end to end delay of reactive protocols by combining properties of both types of protocols. In most of hybrid protocols in VANETs, each node of the network broadcast its routing information (using beacon messages). Thus, nodes create and maintain a table of neighbors based on those beacons. The route discovery is then initialized when a packet needs to be sent and use the neighbors' tables to find destination faster. Hybrid protocols have been made to handle dynamic networks such as Mobile Ad-Hoc Networks

(MANETs) and then modified to fit the high speed and scalability of VANETs. Examples of Hybrid protocols such as Zone Routing Protocol (ZRP) and Temporarily Ordered Routing Algorithm (TORA) are given in more details later in this chapter.

2.3 Categorization of next hop selection

In routings protocols in MANET and VANET, the route discovery phase uses different methods to select the next node in the route (called next hop). Indeed the selection of the next hop is made on several criteria depending on the method used. We distingue height types of next hop selection which are further node, best quality link, most demanding node, probability base, backbone node, stochastic method, counter based, distance to mean. Those method are not necessary independent and are explained below.

2.3.1 Distance-based

The distance-base technique consists of selecting the next hop based on the distance between the current node and the closest to the destination. This technique can either rely on node geographical location especially in VANET but can also rely on the network topology. An important amount of algorithm uses this technique during the route discovery phase named greedy forwarding. In greedy forwarding, each node selection the next hop the closest geographically to the destination (This technique is called furthest node selection). In broadcasting routing protocols where the destination position is not known, each node selects the node the furthest away from itself in its transmitting range considering that close nodes will already have received the message. The furthest node can also consist of a of each node decide by itself whether to forward the message by comparing its distance to the precedent hop and comparing it to a threshold value.

Examples of Routings protocols using greedy forwarding and further nodes are contained in geographical routing detailed in this chapter.

2.3.2 Best Quality Link

Selecting the next hop based on the quality of the link means considering real word channel conditions. This selection may rely on distance with next hop, the received power from beacon packets or other channel criteria. This type of protocols can increase the end-to-end delay because of the information gathering phase. However, it shows a better reliability than furthest node selection that ignores channel condition.

2.3.3 Most Demanding node

The most demanding node method prioritizes certain nodes of the networks according to their locations in the graph or their time to react to previous messages. The idea is to transmit the message to the nodes the most concerned by it. This type of method is mainly used in warning message propagation in VANET to provide security relative information to the network. These type of selection ensure the delivery of important packets to demanding nodes. However, it increases calculation considerably to identify the demanding area and therefore the end to end delay.

2.3.4 Probability based forwarding

Each node will forward the packet depending on a certain probability. This is used to decrease the number of packet rebroadcasting a packet. The assigned probability is either defined or dynamically change depending on the network density or node location. This method is mainly used for some broadcasting protocols where all nodes of the network are concerned by the message so the objective is to decrease the number of rebroadcasting nodes.

2.3.5 Backbone node

Backbone node selection consider the existence of infrastructure in the network. Those can be physical, using road side units for examples like in some trajectory based routing protocols. Certain can also be dynamically created by forming groups of nodes where some of them will be entitled to the routing (like in cluster based routing algorithm). Those infrastructures will be used to either calculate the route to destination or to select next forwarding node. The goals of this selection is to decrease the overhead by allowing only backbone nodes to transmit However, the node selection requires calculation that are needed to be kept as simple as possible to maintain a reliable end to end delay.

2.3.6 Stochastic method

In stochastic method, the next hop is selected randomly among the neighbors available in transmitter based algorithm. In receiver based algorithm, each node selects randomly either to forward or not the packet. This type of selection do not ensure the packet delivery and is therefore not reliable in route request or broadcast routing.

2.3.7 Counter based

The counter based method consists on counting the number of time a message is received by a node. When a node receives a message, it will set up a time to wait before forwarding the message and if it receives again the message before this time is over, it will cancel the forwarding. This defined time is called a back off timer and can be calculated over some parameters such as node location for example. This solution can be most of the time used with another selection technique sur as distance-based for example in the form of a back off timer. The Counter-based method is mainly used in geographic

broadcast routing protocol to propagate message to all node and decreasing overhead. However, it can also be used to create route to destination among the first node to forward to the destination.

2.3.8 Distance to mean

The Distance to mean selection can be compared to the distance-based technique and is mostly a receiver-based method, where each node decides by itself to forward the message. However, here the decision is made on the distance between the node and the spatial mean of precedent forwarders of the packet. This value is then compared to a threshold value calculated over several parameters. This selection method shows better results than distance based ones such as further node in terms of reachability, bandwidth consumption and end to end delay [1]. Therefore, the algorithm proposed on this paper will use this type of next hop selection. Algorithms based on this technic will be detailed in more depth in the next sections of this chapter.

2.4 Types of routing protocols

Now that three types of communication have been distinguee, the routing protocols based on those communication can be separated into six categories. We identify topology based, location based, cluster based, Geocast based, trajectory based and geographic based routing. All those categories are sorted by the way the transmitted message is conveyed to its destination. This literature survey is also based on several surveys such as [2], [3], [4], [5], [6], [7], [8], [9] and [10] and on all the paper referenced in the next section.

2.4.1 Topology Based Routing

Topology based routing consist on the establishment and the maintenance of routing table for each node of the network. It means that every node in the network knows the path to reach other node in the network. This type of protocol is not fitted for VANET because of the high mobility and scalability of vehicular network. Maintaining such table is hard when the topology of a network is constantly changing. However, certain protocol adapts such protocols to make it more efficient in VANET environment.

2.4.1.1 AODV

Ad-hoc On-Demand Vector (AODV) routing [11], [12] is a neighborhood awareness protocol because each node uses Beacon hello messages to keep track of its neighbors. AODV is also a reactive routing protocol because the route generation is started only when a node wants to send a packet. Then, a Route Request (RREQ) is sent to neighbors and propagated between net-hop neighbors to find a path to destination. Then, a Route Reply (RRep) is sent back to the source using the reverse route. AODV control if those routes contain no loop and find the shortest path among them. Each node then stores next hop to destination in a table. It can also create new routes or modify existent by handling errors.

AODV is one of the most used protocol in wireless networks because of its viability, however, it is not fit to handle high mobility and scalability among VANET. Therefore, a lot of protocol used in VANET are adapted from AODV with modification to fit more VANET the needs. Finally, AODV provide Unicast and Multicast by establishing routes to destination which is node needed when you send warning messages in VANET which only needs Broadcasting.

2.4.1.2 MAV-AODV

Multicast with Ant Colony Optimization of AODV (MAV-AODV) [13] is based on AODV and is a Bio inspired algorithm. Based on Ant colony, MAV-AODV use nodes' mobility information to build multicast tree and sustain its lifetime. A neighbor table is maintained using periodic beacons to obtain mobility information. Then the Request Response phase of AODV is improved using node's position and a Best quality link next hop selection.

2.4.1.2 OLSR

Optimized Link State Routing (OLSR) Protocol [14] is a proactive protocol because it creates and maintain a routing table based on topology information regularly exchanged by the nodes. Then certain nodes are classified as Multi-Point Relay (MPR) by their neighbors. Information that they broadcast periodically. Those nodes are then used to form the route from a node to the destination. Like AODV, maintaining a routing table in VANET networks is not efficient because of the ephemeral state of the network but also in terms of bandwidth consumption. Then, OLSR also provide Unicast and Multicast which are not needed for warning message propagation in VANET.

2.4.1.3 DSDV

Destination Sequenced Distance Vector (DSDV) [15] uses routing table scheme where the path is calculated based on the Bellman-Ford algorithm which is, in graph theory, an algorithm to find the shortest path form a single vertex to other vertices in a weighted digraph. The main used of this algorithm is to avoid the routing loop problem.

2.4.1.4 DSR

Like AODV, Dynamic Source Routing (DSR) [16] starts route discovery operation when a node wants to send a message. But unlike AODV, the route is kept in full in the table and maintained for a period. Likewise, the RREQs are made using flooding, not by using a maintained table of neighbors. Furthermore, every node is responsible to wait for reception confirmation from next hop. Until then, it will keep sending the packet until it reaches a defined maximum threshold.

2.4.1.5 TORA

TORA [17] is alike DSR but in addition to the route discovery phase, this protocol provides a route erase phase. Thanks to the first phase, every node constructs and maintains a route table. Nodes are then able to detect network partition, in that case, they will trigger the erase phase by sending a clean packet that will delete the invalid route.

2.4.1.6 FSR

FSR [18] uses flooding broadcasting to propagate packets in the networks. Then, with the latest location information contains in those messages, each node builds and maintains a Topology Table which allows node to build routes.

2.4.1.7 ZRP

In ZRP [19], every node's neighborhood is delimited by a defined zone depending on the transmission range. For nodes inside this area, routes are discovered reactively. However, to transmit to nodes outside of the transmission zone, a route request is emitted to other zones.

ZRP is a hybrid protocol, between topology based and cluster based because nodes are grouped in zone. But in Cluster based routing algorithm a cluster head is designed to deal with all routing inside and outside of the cluster.

2.4.1.8 MAR-DYMO

Mobility-aware Ant colony optimization Routing (MAR-DYMO) [13] is an integration in Mobile Ad-Hoc Networks (MANET) of Ant Colony Optimization (ACO) [20] by combining it to the routing protocol Dynamic MANET On-demand (DYMO).

DYMO is the successor of AODV protocol and is based on the same principle of multi-hop propagation between neighbors until it reaches destination. ACO works with several principles. One of them is the pheromones which are used to grade route to increase reliability. In MAR-DYMO, more pheromones are added on RReq route. Then if a RRep crossed the same route, more pheromones are added. Route is then selected according to pheromones density. Furthermore, pheromone evaporates with time and are added by transmitted packet to maintain and modify route if needed. MAR-DYMO also uses a Kinetic Graph framework [21] to make prediction about node's neighbors trajectory. It uses aperiodic HELLO message sending compared to DYMO and reduce Bandwidth consumption.

2.4.1.9 QoSBeeVANET

Quality of Service Bee Swarm routing protocol (QoSBeeVANET) [22] is a topology based protocol designed for unicast routing inspired over bee swarm. In this protocol, the first phase which is route request (RReq) is implemented using stochastic broadcasting. Which means, every node of the network is given a random or determined probability to forward a message. This type of broadcasting helps reducing the number of

bandwidth consumption. As soon as the destination is reached a RRep is sent back to the source and the route is stored in a routing table with the following information: next hop, number of hop before destination, hop count. The algorithm maintains routes by sending periodically packets to neighbors and if it detects a missing node or a loss of Quality of Service (such a too high bandwidth consumption or end to end delay), it will warn all other nodes concerned on the degraded path. All nodes on error are removed and a new route discovery phase is started. After all, this algorithm can easily flood the network because of the number of packets send, especially if an error occurs (due to node missing or QoS not respected).

2.4.1.10 HyBR

Hybrid Bee Swarm Routing (HyBR) [23] has been designed to overcome drawbacks faced by QoSBeeVANET. Still designed to propose unicast and multicast routing, HyBR use two types of routing depending on the density of the network. Topology based routing when the density is high and geography-based routing when the density is low. The topology based routing RReq and RRep is executed like in QoSBeeVANET using stochastic flooding for RReq. The RRep is then routed back to destination throughout discovered path and stored in a table.

The geography-based routing is based on shortest node algorithm. A broadcasting flooding is executed to determine all routes to destination. Then the route is selected hop by hop with the node the closest in hop distance to the destination. Results obtained by this algorithm are close to those obtained with AODV and

GPSR.

2.4.1.11 Datataxis

Datataxis [24] is a topology based unicast routing protocol inspired by the behavior of Bio-System: *Escherichia coli* bacteria (an active component of in the natural immune system). Datataxis is designed to collect metadata (such as location, time stamp, etc.) in urban environment. Then those data were proposed to be diffused using the protocol MobEyes. Datataxis estimate firstly the meta-data density by road segment and then send multi-agents systems allowed to move from nodes to nodes to collect those data. This protocol has been proposed essentially for distributed surveillance and monitoring, for police car for example. Therefore, we will not detail it in more detail.

2.4.1.12 MURU

Multi-hop routing for Urban VANET [25] is a topology based routing protocol designed for unicast and multicast communication. The protocol is based on AODV but instead of using hop count to find optimal path to destination, the Expected Disconnection Degree (EDD) is used. EDD is calculated over the Packet Error Rate (PER) of link. PER being function of hop distance, EDD being the probability that a link break then depends mainly on hop distance. EDD depends also independently to predicted speed, movement trajectory and vehicle location. Knowing that in the paper, the vehicle's mobility is approximated to a first order Markov chain. The Markov chain define a stochastic process in which the conditional probability distribution of future state depends only on the current state.

Besides this improvement, the first phase of the protocol, the route request broadcasting is constrained by vehicle movement trajectory.

This protocol shows good result but the number of information required to calculate EDD can be difficult to obtain in real life scenario. Furthermore, as other topology protocol, maintaining path topology decrease scalability of the protocol.

2.4.2 Position Based Routing

The lack of scalability and robustness of topology-based protocol has lead research to find other type of routing protocols. Indeed, the creation and maintaining of routing table in highly scalable networks may not be reliable. Position-based routing protocols which use network location of nodes to decide how to route messages are a new area of research. Although, Geocast routing can be defined as a position based routing protocol because it defines area based on geographical coordinates where nodes are concerned about the message.

2.4.2.1 GPSR

Greedy Perimeter Stateless Protocol (GPSR) [26] uses periodic beacon messages to build neighbors table on each node. The next-hop selection is distance based and uses GPS node's location. GPSR also integrates recovery strategies based on perimeter routing to eliminate redundant routes. This protocol is the most used in VANET to run simulation because it presents good reachability and end-to-end delay.

2.4.2.2 AMAR

Adaptive Movement Routing (AMAR) [26] like GPSR, uses a greedy forwarding technique for next hop selection. But in addition to the distance, AMAR also use neighbors' position, direction and speed to select the next hop.

2.4.2.3 GYTAR

Like precedent protocol, GreedY Traffic Aware Routing (GYTAR) [26] bases its next hop selection on greedy forwarding. GYTAR also use periodic beacon to maintain a table of neighbors containing position, velocity and direction. Secondly, it defines junction based of nodes density close to the node. GYTAR uses then the table and the junction density to select next hop between its neighbors.

2.4.2.4 DREAM

Distance Routing Effect Algorithm Mobility (DREAM) [27] acquires each node's position using local services. It then calculates the possible destination area position and use directional flooding to reach it. The directional flooding consists of restricting the flooding graph to nodes in the area that leads to the destination.

2.4.2.5 LABAR

Location Area Based Ad-hoc Routing (LABAR) [27] uses a backbone next hop selection using V2*I* communication (with road side unit) to create an infrastructure in the network. LABAR then routes message from mobiles nodes using fixed backbone nodes. To determine the route among fix nodes it uses directional routing such as AODV.

2.4.2.6 ROVER

Robust Vehicular Routing (ROVER) [28] represents an example of Geocast routing. Geocast routing consider that only certain area is concerned by the message sent. Those area are called Zone Of Relevance (ZOR) and are defined by their GPS location. Each packet is then affected to one ZOR and will be forwarded to each node in this area. Each node discovers in which ZOR it belongs using local services.

2.4.2.7 pPSO (AODV)

The parallel Particle Swarm Optimization (pPSO) [29] for VANET is inspired by swarm and is a parametrization of the protocol AODV. It consists of calculating the best position and speed for the vehicles to occupy in order to make the protocol AODV reduce its packet overhead, end-to-end delay and delivery ratio.

2.4.2.8 GSR

Geographic Source Routing (GSR) [30] uses a Reactive Location Service (RLS) and a digital map to handle routing in urban area. With the location of the destination acquired with RLS and fixed node in the network (RSU at intersections), GSR use Dijkstra to calculate the shortest path between junction (fixed nodes) and greedy forwarding to disseminate the packet.

2.4.2.9 CAR

Connectivity-Aware Routing (CAR) [31] uses neighbor recognition using beacon messages sent with a time interval depending on the number of neighbors detected. The route request phase use anchor points selected over best quality link method. Then the packet is forwarded using a greedy forwarding method among those anchor points.

2.4.3 Cluster Based Routing

Cluster based routing consists of dividing the network into smaller groups of nodes. Among each group a cluster head will be selected, basically a node will handle every communication. The ones inside the cluster but also the one outside it. The main objective of cluster based routing is to handle high scalability of VANET by handling

smaller connected infrastructure networks. Although, this type of protocol often uses GPS information to delimit and organize its cluster.

2.4.3.1 CBLR

The Cluster Based Location Protocol (CBLR) [9] builds its cluster using bacon hello messages as an initialization phase. This operation is also used to define the cluster head that will construct and maintain a table containing information over nodes in the cluster and others cluster heads of the network. Dissemination inside the cluster is effected using a greedy forwarding techniques. Communication outside of the cluster is handled firstly by finding the location of the destination using other cluster head information

2.4.3.2 CBDRP

Cluster Based Directional Routing Protocol (CBDRP) [32] builds its cluster on nodes velocity vector (speed and direction). The routing is then effected like in the CBLR protocol.

2.4.3.3 EDCBRP

Euclidian Distance Cluster Routing Protocol (EDCBRP) bases the network division in cluster on Euclidian distance. Nodes with a Euclidian distance with each other under a defined threshold form a group. The distance is calculated using GPS information of nodes and acquired by hello message beacon periodically send. Topology tables are maintained inside clusters. For communication with other cluster, a route-request route response is sent in order to build the route to destination.

2.4.3.4 TACR

Trust Dependent Ant Colony Routing (TACR) [33] is a Bio inspired routing algorithm for VANET. In TACR, cluster are builds on position and speed of moving nodes in the network. The cluster head is selected on lowest node speed with priority to RSU because of their fixed position and infrastructure network available. The communication between cluster is achieve using the Ant Colony Optimization algorithm. It consists of a route request sent to every other cluster that check if the destination is in its member table. If yes, the cluster head do not forward the route request but instead, send a route response backward. Inside cluster communication are maintained with maintaining a member table using Beacon messages.

2.4.4 Trajectory Based Routing

Trajectory based routing are developed mainly for urban environment with Road Side unit (RSU) positioned on roads intersection. Those routing protocols propose to use a fixed infrastructure composed by the RSUs and disseminate message to moving node using trajectory calculation. This type of algorithm modifies the V2*I* and I2V communication by adding several information transmitted to and by fixed RSU. Thereby, traffic statistics such as density, average speed, average direction or digital map of the area can be transmitted over V2I.

2.4.4.1 AMR

The Adaptive Message Routing (AMR) [34] algorithm has for main objective to reduce the end-to-end delay. To achieve its goal, it builds route using a genetic algorithm based on the probability of connectivity and the hop count. AMR uses backbone next hop selection, using fixed RSU to convey messages over the network. The algorithm

calculates the intersection between the source and the destination and the infrastructure network build on RSU.

2.4.4.2 IGRP

The Intersection-based Geographical Routing Protocol (IGRP) [35] is mostly used to send packet from vehicles to the internet using a genetic optimization algorithm over intersection routing protocol. This algorithm uses a backbone next hop selection technique among RSUs until it reaches an internet point.

2.4.4.3 TBD

The Trajectory Based Data (TBD) [36] algorithm uses vehicular density, mobility patterns, average speeds and digital map transmitted over V2I communication to evaluate its best next-hop to reach the closest RSU with the lower end-to-end delay. Then, it shares this delay with close nodes for them to build their own path to RSU. Packets are sent over RSU network that will disseminate it to destination using classical infrastructure networking routing.

2.4.4.4 TSF

Trajectory-based Statistical Forwarding (TSF) [37] calculate end-to-end delay in the opposite way of TBD, meaning from the fixed node to the moving vehicles. Then based on the minimal delay between nodes and RSU, the route between source and destination is calculated with destination trajectory. An optimal target point is identified between the destination node's trajectory and an RSU on the network. Therefore, the packet will be forwarded over the RSU network to reach the optimal target point at the same moment as the destination node.

2.4.4.5 TMS

Trajectory-based Multi-Anycast (TMS) [38] assumes the existence of a Traffic Control Center (TCC) containing information of all nodes in the network (position and velocity). Based on those information, every time a packet is sent, the TCC identify a rendezvous point between destination node's trajectory and a forwarding tree build on moving nodes.

2.4.4.6 STDFS

Shared-Trajectory-based Data Forwarding Scheme (STDFS) [39] uses RSU to propagate nodes' trajectory over the network. With those information, every node can calculate a rendezvous point with the destination and disseminate the packet over V2V communication to the target point.

2.4.5 Geographic and Broadcast Routing

Broadcast routing protocol are mainly used to transmit warning information or other data that concern every vehicle on the road. However, this type of routing is also used as the route request phase for some multi-cast or unicast protocols. For the route discovery phase for example. The main goal of such routing is to reduce bandwidth consumption by skipping the route discovery phase and therefore, decreasing the number of routing packet sent on the network.

2.4.5.1 Hybrid-DTN

Hybrid geographic and Delay Tolerant Networks (Hybrid DTN) [40] uses a greedy forwarding with a furthest node next hop selection as a route discovery phase.

However, if the greedy forwarding fail, it uses the perimeter forwarding (or right-hand rule) method to reach the destination. This method consists for a node, of covering a counterclockwise circle around itself and forwarding the packet to its first neighbor found in this circle.

2.4.5.2 SRB

Secure Ring Broadcasting (SRB) is based on the best quality link selection. Indeed, it classify nodes based on receiving power to estimate the distance between the receiver node and the last broadcasting one. Then only those at the preferred distance from the last broadcasting node can forward the packet several times. This algorithm is based on a flooding techniques. However, depending on the estimated distance between graph level, nodes can forward one or several times.

2.4.5.3 PBSM

Parameter less Broadcast in Static to highly Mobile (PBSM) only checks if nodes in the neighborhood has received the packet and then retransmit the packet only to those who did not receive the packet.

2.4.5.4 EAEP

Edge-Aware Epidemic Protocol (EAEP) uses probabilistic techniques to decide whether to forward the packet or not. A time probability is calculated by each node to decide when to forward the packet.

2.4.5.5 DV-CAST

Distributed Vehicular broadcast (DVCAST) [41] uses periodic beacon to maintain a table of neighbors for each node to know local connectivity. Then, depending on the

connectivity of each node, action of forwarding is decided. DV-CAST sort the node in two categories, the well-connected ones and the sparely connected ones. The next forwarding node in the well-connected one is selected over the distance with the sender node. Indeed, a back off timer is calculated inversely proportional to the distance with the sender. The node with the smaller back off time (the furthest node) will then rebroadcast the message. For the second category, the sparely connected ones, if a node has a neighbor in the opposite direction road it will rebroadcast immediately, if not, then it will keep the packet until it finds another node in the opposite direction road.

2.4.5.6 TRADE

TRAck DEtection (TRADE) protocol gives nodes a table of neighbors maintained using periodic beaconing. A node sorts its neighbors in three categories depending on their position and velocity: same road ahead, same road behind and different road. Then it uses the furthest node selection technique as next hop selection in the two first categories. For the third one, the node just rebroadcast to every node in this category.

2.4.5.7 MAC

The Media Access Control (MAC) [42] protocol also uses periodic beaconing to maintain a table of neighbors on each node. Consequently, every node calculates its relative direction to the sender and the ones between the sender and its neighbors. Then if the node's relative direction corresponds to the packet direction, the node will check in its neighbor table if it is the more dedicated to forward. This selection is based on the furthest node criteria using the segment define by the distance and the relative direction calculated before.

2.4.5.8 REAR

Reception Estimation Alarm Routing (REAR) protocol is based on probabilistic next hop selection among neighbors in the direction of the message propagation. REAR makes node maintain a neighbor table and each warning message sent contains sender position and a list of sender's neighbors. Each node in the direction of the message's direction will then calculate its reception probability and wait a back off time inversely proportional to this value. The first node to reach its back off time will then rebroadcast the message.

2.4.5.9 GPCR

Greedy Perimeter Coordinator Routing (GPCR) [43] defines junction as a link between one node reached by a message and another one out of the message's range. It uses 2-hop neighbors table to locate a junction between to nodes. Indeed, by sending periodic beacon containing its position and its neighbors, each node can construct such a table. Therefore, if a node has two neighbors that does not have each other on their respective table, that means this node is a junction. Every node calculates the number of junction it represents and the those that have the bigger coefficient are called coordinator. Coordinators informs other nodes of their new role and form the backbone of the network. Consequently, the next forwarding hop selection during the broadcasting will prefer a backbone node as the next forwarding node. If no coordinator is found, the furthest node method is used.

2.4.5.10 TDR

Three-Dimensional scenario oriented Routing (TDR) [44] protocol proposes an improvement of GPCR protocol and its upgrade GyTAR to fit 3-Dimensional environment. The main difference comes from the size of hello beacons which contain 3 coordinates instead of two.

2.4.5.11 Multicast Routing for Message dissemination protocol

This protocol also uses beaconing to maintain neighbor's awareness table and the next hop selection is based on the most demanding node criteria.

2.4.5.12 OAPB

Optimistic Adaptive Probabilistic Broadcast (OAPB) [45] constructs and maintains for each node a 2-hop neighbors table using periodic beacon sending. Then each node calculates a probability of rebroadcasting based on that information. it create a back off timer, function of the probability of rebroadcasting and a random variable. The node with the smallest back off timer will be selected as the next hop for the message.

2.4.5.13 UMB

Urban Multi-Hop Broadcast (UMB) protocol considers that every message should be transmitted to RSU disposed to every intersection and that should behave as repeater. If there is any RSU available, the algorithm will function on a reactive way. When message needs to be sent it will send a directional request to relay (RTS) containing its position and direction of propagation. The nodes in the broadcast area defined by the packet direction emit a signal of a duration proportional to the distance with the sender and the number of jamming signal in its transmission range. The node with the longest

jamming (i.e. the furthest away from the sender) signal will send the Clear To Broadcast answer (CTB) to the sender and will be designed as next hop. This method can be classified as furthest node selection. When the warning message is sent, elected relay will send back an ACK message to the sender. If it fails to do so or if several CTB are received, UMB will start the recovery algorithm.

2.4.5.14 SB

Smart Broadcast (SB) is based on UMB but replaces the jamming signal of the furthest node selection method by a classic back off timer inversely proportional to the distance with the transmitter. This way the next relay is not the one that wait the most of time like in UMB. SB also handle CTB collision better with a random selection between the two possible routes.

2.4.5.15 STAR

Intersection based routing protocol [46] that uses RSU placed at red light intersections. Packet are forwarded on every red light connected and to every first car on green light that are the closest to the destination.

2.4.5.16 VanetDFCN

VANET Delayed Flooding with Cumulative Neighborhood (VANETDFCN) [47] is an improvement of the protocol DFCN for MANET. It transmit more information about neighbors (such as the position) to improve the forwarding node selection method. The next hop selection is distance based, if the distance between the receiver and the sender is over a Distance To Live, the node will not forward. The packet must also have been

received only once. Based on those criteria, a coefficient is calculated which represent the number of TCP chunk a packet can be divided in and transmitted on the liaison created.

2.4.5.17 xChangeMobile

xChangeMobile [48] is an improvement of VanetDFCN with the addition of a Threshold on the minimal number of chunk that can be transmitted over a liaison. Only node that can provide a liaison with a coefficient higher than the Threshold will be considered as forwarding node.

2.4.5.18 MOCell

MOCell [49] routing protocol is based on a genetic algorithm to improve xChangeMobile. The goal is to reduce bandwidth consumption and the number of lost chunks in the TCP process.

2.4.5.19 RBLSM

Reliable Broadcasting of Life Safety Messages (RBLSM) is also a reactive protocol that sends RTS and waits for CTB when a packet needs to be sent. However, RBLSM uses the nearest node next hop selection instead of the furthest.

2.4.5.20 LW-RBMD

Light Weight Reliable Broadcast Message Delivery (LW-RBMD) [50] protocol uses the furthest node technique using only the header of warning messages to transmit sender's position. Nonetheless, the sending node will wait for a rebroadcasted message and take it into an acknowledgment (ACK). If the ACK is not received, the sender will

resend the packet after a timer. This protocol is designed to limit the network overhead and still maintaining high reliability.

2.4.5.21 MHVB

Multi-Hop Vector Broadcasting (MHVB) is a broadcasting routing algorithm using the furthest node selection technique with a classical inversely proportional back off timer set up. However, MHVB integrates a congestion detection algorithm, which consist of detecting when the number of neighbors is above a certain Threshold. In that case, the Broadcasting interval is increased to lower the bandwidth consumption.

2.4.5.22 DTM

Distance-To-Mean (DTM) [51] algorithm uses the distance to mean as next hop selection technique. Each node maintains a table of neighbors using beaconing and defines a distance threshold based on the number of neighbors. This distance represents the minimum the distance to mean value must exceed so that the node is considered as a possible forwarding node. Firstly, introduced in 2011, it presents better results than distance based greedy forwarding in terms of reachability and of network overhead. This paper being based on this paper, DTM will be explained in more details in the next section.

2.4.5.23 DADCQ

Distribution-Adaptive Distance with Channel Quality (DADCQ) [52] protocol uses both distance, and best quality link as next hop selection. Here, warning messages also propagate their neighbor table in their header. This way, each node has access to two-hop neighborhood.

2.4.5.24 CSBD

SCBD [53] is a MAC and network cross layer with density-adaptive contention window. This means that the MAC layer is directly influenced by the information obtained on the network layer using geographical routing and distance-to-mean heuristic. [51].

2.4.5.25 SLAB

Statistical Location Assisted Broadcast (SLAB) [54] enhances DADCQ using the distance-to-mean next-hop selection method instead of the distance-based used in DADCQ. Also, SLAB uses machine learning techniques to improve the Threshold function definition.

2.4.5.26 FLB

The Fuzzy Logic Based (FLB) [55] protocol is based on the DTM algorithm but uses fuzzy-based techniques to calculate the threshold value. Indeed, each node is classified according to several factors defined on its information such as velocity, position and number of neighbors; the Coverage factor, the Mobility Factor and the Connectivity Factor. This improvement in the broadcasting node selection presents better results than DTM in terms of number of rebroadcast per covered nodes.

2.4.5.27 BEFLAB

Bandwidth Efficient Fuzzy Logic-Assisted Broadcast (BEFLAB) [56] for VANET, like FLB presents a Fuzzy-logic based receiver based rebroadcasting node selection. Each node is here classified according to two factors, the Mobility Factor and the Coverage factor. Then a table of Fuzzy rules decide whether a node will rebroadcast

or not. Unlike FLB, BEFLAB does not use the distance-to-mean heuristic method but only the fuzzy technique.

2.4.5.28 IHAB

Intelligent Hybrid Adaptive Broadcast (IHAB) [57] for VANET is based on FLB and BEFLAB. Indeed, IHAB first calculate the node potential transmit density (PTD) using two-hop neighbor's information. Then IHAB chooses between FLB and BEFLAB which protocol to use. If the network is said dense (PTD superior to a threshold), IHAB will chose to use BEFLAB, if the network is sparse (PTD inferior to threshold), IHAB will chose to use FLB.

2.5 Conclusion

This literature survey is an overview of routing protocols in VANETs. Table 1 summarizes and classifies all the algorithms presented in this chapter. The current state of the art presents geographical broadcasting routing protocols as ideal for warning message propagation, but the DTM algorithm presents better results than the distance based (or greedy) method, which is the most used technique for next hop selection. However, the overhead generated by beacons represents a disadvantage for DTM based algorithms, compared to immediate broadcasting algorithms. Therefore, we propose the Kinetic Distance-to-mean algorithm.

Table 1: Summary of all protocols described in the literature survey.

Protocols	Reactive - R Proactive - P Hybrid - H	Distance/greedy - G Best Quality - Q MostDemanding - M Probability - P Backbone - B Stochastic - S Counter - C Distance-to-mean - D	Topology - T Position - P Cluster - C Trajectory - Tra Geographic - G	Unicast - U Multicast - M Broadcasr - B
AODV	R	G	T	M, U
OLSR	P	B	T	M, U
DSDV	P	/	T	M, U
DSR	R	flooding	T	M, U
TORA	R	flooding	T	M, U
FSR	P	flooding	T	M, U
ZRP	H	/	T, C	M, U
MAR-DYMO	R	G	T	M, U
QoSBeeVANET	P	S	T	M, U
HyBR	P	flooding	T, G	M, U
Datataxis	P	/	T	U
MURU	R	Q, G	T	M, U
GPSR	H	G	P	M, U
AMAR	H	G	P	M, U
GYTAR	H	G	P	M, U
DREAM	P	flooding, M	P	M, U
LABAR	R	B	P	M, U
ROVER	P	G	Geocast	M, U
AMR	P	B, P	Tra	M, U
Hybrid-DTN	R	G	P	B
pPSO (AODV)	R	G	T	M, U
GSR	P	G	Dijksra	M, U
CAR	H	Q, B, G	P	M, U
CBLR	H	G, B	C	M, U
CBDRP	H	G, B	C, G	M, U
EDCBRP	H	G	C	M, U
TACR	H	G, Q	C	M, U
IGRP	R	B	Tra	M, U
TBD	H	B	Tra	M, U
TSF	P	B	Tra	M, U
TMS	P	B	Tra	M, U
STDFS	R	B	Tra	M, U
SRB	R	flooding, Q	G	B
PBSM	R	/	G	B
EAEP	R	P	G	B

Protocols	Reactive - R Proactive - P Hybrid - H	Distance/greedy - G Best Quality - Q MostDemanding - M Probability - P Backbone - B Stochastic - S Counter - C Distance-to-mean - D	Topology - T Position - P Cluster - C Trajectory - Tra Geographic - G	Unicast - U Multicast - M Broadcasr - B
DV-CAST	H	Q, C, G	G	B
TRADE	H	G	G	B
MAC	H	G, Direction	G	B
REAR	H	P	G	B
GPCR	H	G	G	B
TDR	H	G	G	B
Mult*i* Routing	H	M	G	B
OAPB	H	P, C	G	B
UMB	R	M, G	P	B
SB	R	M, C, G	P	B
STAR	R	B	Tra	B
VanetDFCN	R	G, C	G	B
xChangeMobile	R	G, C	G	B
MOCell	R	G, C	G	B
MAV-AODV	H	Q	T	M, U
RBLSM	R	G	G	B
LW-RBMD	R	G	G	B
MHVB	R	G	G	B
DTM	R	DTM	G	B
DADCQ	R	G, Q	G	B
CSBD	R	DTM	G	B
SLAB	R	DTM	G	B
FLB	R	DTM	G	B
BEFLAB	R	DTM	G	B
IHAB	R	DTM	G	B

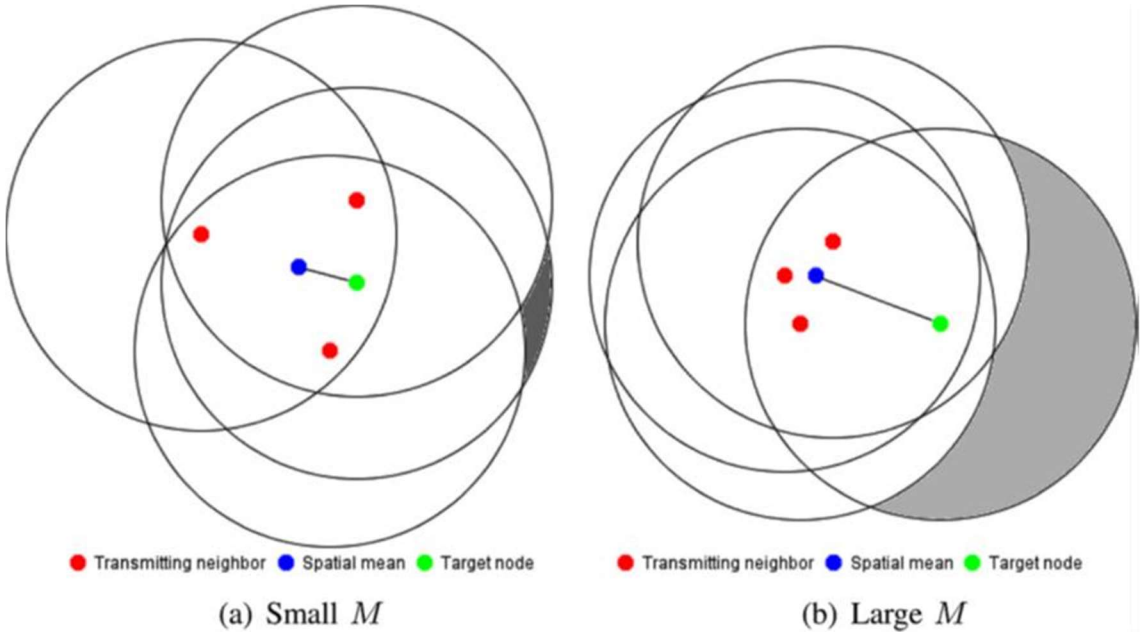
Chapter 3: VANET Kinetic Distance-To-Mean Algorithm (KDTM)

As described in the previous section, broadcasting routing protocols aim to reduce bandwidth and end-to-end delay by not using headshaking mechanisms. Furthermore, direct rebroadcasting seems to be a good type of algorithm for safety message dissemination, because of the low amount of calculation needed and the low bandwidth consumption. However, the distance to mean DTM algorithm presents better results in terms of number of rebroadcasts per covered node than the distance based selection, which is normally used in immediate rebroadcasting algorithms [51] and [1]. Nevertheless, DTM needs beaconing messages to maintain its neighbor table and calculate its threshold value. This paper is therefore about reducing the bandwidth consumption of the DTM algorithm by using a Kinetic graph model and by decreasing the number of beacons sent. In the next subsection, we will firstly detail the DTM algorithm. Secondly, we will explain the Kinetic Graph model. Finally, we will present the Kinetic Distance-to-Mean algorithm based on these two algorithms.

3.1 Distance-To-Mean Algorithm (DTM)

In [51], the authors introduce the Distance-To-Mean statistical method to decide whether a node should rebroadcast or not. Figure 6 from [1] presents the logic behind the distance to mean heuristic method. Here, the receiver point (in green) receives a packet from the three transmitting neighbors (in red). The receiver calculates the spatial mean (in blue) and then its Euclidian distance to it. Finally, if this value is close enough to its transmission range, it means that the new covered area increases consequently the area already covered by the transmitting neighbors. The forwarding is then considered. In Fig.

6, the distance to mean on the left is small compared to the transmission range of the node, therefore the new covered area is narrow. On the right schema, on the other hand, the new transmission range (in gray) is more important due to the bigger distance to mean. The broadcast is then more interesting on the right schema than on the left one.



(a) Small M (b) Large M

Figure 6: Illustration of DTM heuristic from [1].

Therefore, a threshold value is defined to decide when forwarding is interesting. This value is calculated over the neighbors' density. A good threshold value has been found through experimentation in [51] and is given in Eqn. (1), where N is the number of neighbors of the node.

$$Mc(N) = 0.95 - 0.74e^{-0.13N} \quad (1)$$

The calculation of the distance to mean follows several steps. The first one is the calculation of the spatial mean coordinates of the transmitting nodes. The calculation is given in Eqn. (2), where the couple (x_i, y_i) is transmitting node$_i$'s coordinates and n is the number of transmitting nodes.

.

$$(\bar{x}, \bar{y}) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i, \frac{1}{n} \sum_{i=1}^{n} y_i \right) \quad (2)$$

The next step is to calculate the distance between the receiver node and the spatial mean $(\bar{x}, \bar{y})$.This distance to mean *M* for the receiver node at the position (*x,y*) with a transmission radius *r* is given by Eqn. (3).

$$M = \frac{1}{r} \sqrt{(x - \bar{x})^2 + (y - \bar{y})^2} \quad (3)$$

M is then compared to the threshold value *Mc*: if *M* is superior to *Mc*, then the node will rebroadcast.

To calculate the spatial distribution, several messages must be received. Therefore, a back off timer mechanism is implemented and set inversely proportional to the distance to mean value *M*. This is done to favorize the node with the highest distance to mean, so that it rebroadcasts first. The time calculation is shown in Eqn. (4), where $T_{\max}$ is the maximum time value and *M* is the distance to mean.

$$T = (1 - M)T_{\max} \quad (4)$$

In the end, the final algorithm follows those steps:

1. Beacon messages are sent periodically to create and maintain a neighbor table
    a. Based on this table we calculate the number of neighbors *N*.
    b. Calculate *Mc* using Eqn. (1).
2. Initialize $(\bar{x}, \bar{y})$
3. When a message is received:
    a. update $(\bar{x}, \bar{y})$ using Eqn. (2).

b. Calculate *M* with Eqn. (3).

c. If *M* inferior to *Mc*, interrupt the procedure

d. If *M* superior to *Mc,* calculate and set a back off timer *T* using Eqn. (4)

4. If a message is received during the back off time *T*, repeat 3

5. When the back off time is over, rebroadcast if *M* superior to *Mc.* Figure 7 presents the DTM algorithm that regroups all those steps.

Algorithm 1 DTM algorithm

```
1. Send Beacon message and maintain table 2. Initialize Mc
using Eqn. (1).
3. Initialize (x̄, ȳ).
4. While Receiving message do
5.     Update (x̄, ȳ) using Eqn. (2)
6.     Calculate M using Eqn. (3)
7.     If M < Mc do
8.         interrupted the procedure
9.     If M > Mc do
10.        Set back of timer T using Eqn. (4)
11. end while
12. If Message receive during T do
13.     Repeat 4
14. else
15.     If M < Mc do
16.         interrupted the procedure
17.     If M > Mc do
18.         Rebroadcast message
```

Figure 7: DTM algorithm.

This algorithm is the concept of DTM, but to maintain the neighbor table up to date, the beacon messages that are used occupy a good part of the bandwidth consumption. The Kinetic Graph model proposed in [21] presents a way to reduce the overhead of beaconing in VANET protocols.

3.2 Kinetic Mobility Management Applied to Vehicular Ad Hoc Network.

The Kinetic Model proposed in [21] comes up with a way to reduce the number of beacons sent to maintain the neighbor table by predicting the time a vehicle joins and leaves a neighborhood. Knowing those parameters, the hello messages will not be sent periodically but instead, sent at a chosen time.

3.2.1 Kinetic Node Degree

The first step proposed by the authors is to set weights on links between the receiver and its neighbors. Several weights are first introduced and then combined to define a coefficient to each link, which represents its reliability.

Firstly, we define the power cost function as the amount of power required to transmit on a link between the node *i* and its neighbor *j*. We use the distance between these nodes to represent this function. We defined D_{ij} in Eqn. (5) as a function of the time *t* and the distance between *i* of position (x_i, y_i) and *j* of position (x_j, y_j). A_{ij}, $B_{ij}$and C_{ij} are three constants defined function of position of nodes *i* and *j*.

$$D_{ij}^2(t) = D_{ij}^2(t) = \left\|\mathrm{Pos}_j(t) - \mathrm{Pos}_i(t)\right\|^2$$
$$= \left(\begin{bmatrix} x_j - x_i \\ y_j - y_i \end{bmatrix} + \begin{bmatrix} dx_j - dx_i \\ dy_j - dy_i \end{bmatrix} \cdot t\right)^2 \quad (5)$$
$$= A_{ij}t^2 + \mathrm{B}_{ij}t + \mathrm{C}_{ij}$$

Considering r as the maximum range of transmission, we can find t_{ij}^{from} and t_{ij}^{to}, the time when j enters the neighborhood of i, and the time when it exits the neighborhood, respectively by solving Eqn. (6) below.

$$D_{ij}^2(t) = r^2 <=> A_{ij}t^2 + B_{ij}t + C_{ij} - r^2 = 0 \quad (6)$$

The next proposed weight is the probability $p_{ij}(t)$ of a link between the node i and j to be maintained, depending on their respective probability to continue their trajectory ($p_i(t)$ and $p_j(t)$ respectively). We define:

$$p_i(t) = e^{-\beta_i(t-t_i)} \quad (7)$$

Where:

- t_i is the time when the node i started its trajectory
- $\frac{1}{\beta_i}$ corresponds to the Poisson parameter that is the average time a node stays on the same trajectory

Using Eqn. (7) and by assuming trajectories of nodes i and j independent we define:

$$p_{ij}(t) = p_i(t) \cdot p_j(t) = e^{-(\beta_i+\beta_j)\cdot\left(t-\frac{t_i\beta_i+t_j\beta_j}{\beta_i+\beta_j}\right)} = e^{-\beta_{ij}\cdot(t-t_{ij})} \quad (8)$$

$p_{ij}(t)$ is the probability of i and j continuing their trajectory, we call it stability of a link. Figure 8 represents the stability of the link ij for a defined β_{ij} and t_{ij}. The shape of the curve shows that the stability of the link ij decreases when the time increases.

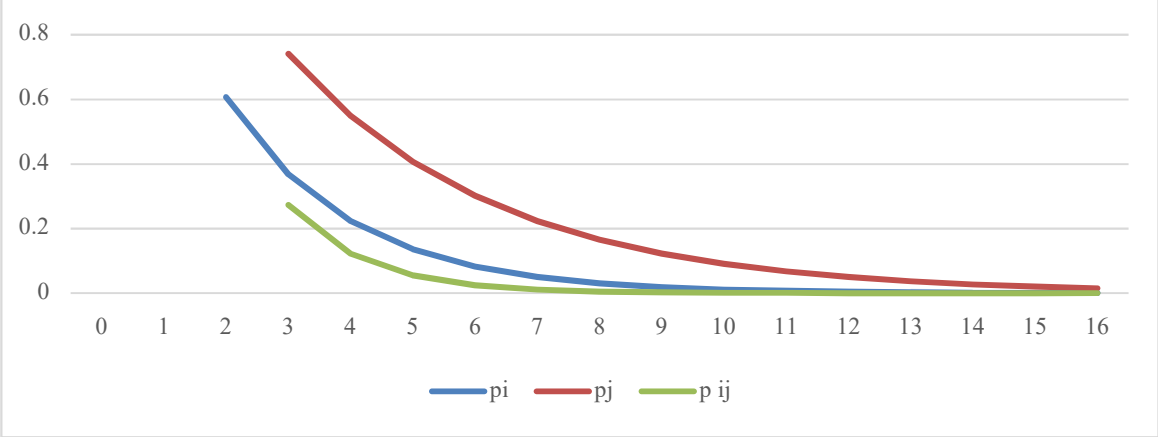


Figure 8: Stability of link *ij*.

The next parameter defined in [21] is the degree of a node. It represents the number of neighbors a node has at the instant *t*. Basically, it is the product of two sigmoid, defined over the times t_{ij}^{from} and t_{ij}^{to}, solution of Eqn. (6) for each neighbor *j* of a node *i*.

Equation (9) shows the expression of a Sigmoid:

$$sigm_{t_i}(t) = \frac{1}{1 + \exp(a \cdot (t - t_i))} \tag{9}$$

Equation (10) shows the degree of the activation of the link *ij* between t_{ij}^{from} and t_{ij}^{to} with the product of the respective sigmoid. Figure 9 presents the degree of a link *ij* between the time *t*=2 and time t=6.

$$deg_{ij}(t) = \frac{1}{1 + \exp\left(-a \cdot \left(t - t_{ij}^{\text{from}}\right)\right)} \cdot \frac{1}{1 + \exp\left(a \cdot \left(t - t_{ij}^{\text{to}}\right)\right)} \tag{10}$$

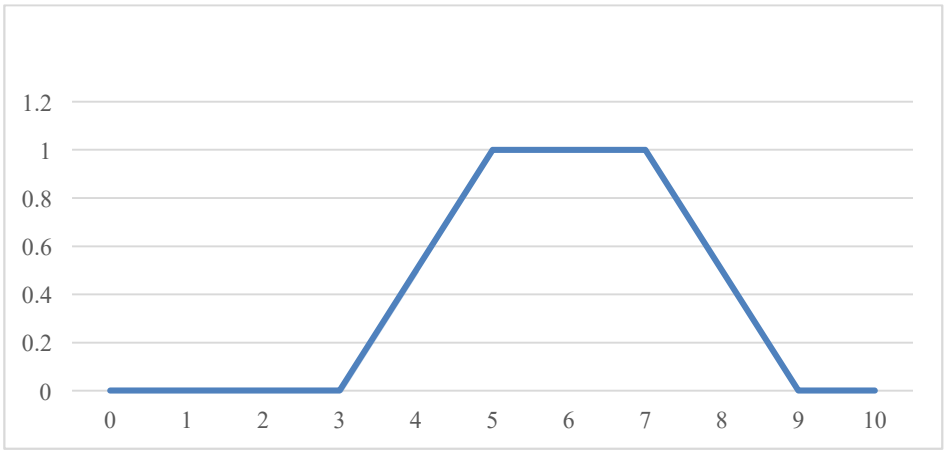


Figure 9: Degree of the link ij between $t_{ij}^{\text{from}} = 4$ and $t_{ij}^{\text{to}} = 8$.

Then, Eqn. (11) presents the degree of the node *i*, sum of all the activated link with its neighbors *j,* function of time. $neighbors_i$ is the number of neighbors of *i* at a time *t.*

$$Deg_i(t) = \sum_{k=0}^{neighbors_i(t)} \left(\frac{1}{1 + \exp(-a \cdot (t - t_{ik}^{\text{from}}))} \cdot \frac{1}{1 + \exp\left(a \cdot (t - t_{ik}^{\text{to}})\right)} \right) \quad (11)$$

Finally, we define the Kinetic degree as the probabilistic degree at the time *t*, calculated by multiplying the activity $Deg_{ij}(t)$ of a link by its stability $p_{ij}(t)$. Equations (8) and Eqn. (11) thus imply Eqn. (12), showing $\widehat{Deg_i}(t)$:

$$\widehat{Deg_i}(t) = \int_t^{\infty} \left(\frac{1}{1 + \exp\left(-a \cdot \left(t - t_{ik}^{\text{from}}\right)\right)} \cdot \frac{1}{1 + \exp\left(a \cdot (t - t_{ik}^{\text{to}})\right)} \cdot e^{-\beta_{ij} \cdot (t - t_{ij})} \right) \quad (12)$$

Figure 10 compares the kinetic degree and the degree of a node *i* for which nodes *j*, *k* and *l* enter the neighborhood from the respective time $t_{ij}^{\text{from}} = 2$ to $t_{ij}^{\text{to}} = 12$, $t_{ik}^{\text{from}} = 5$ to $t_{ik}^{\text{to}} = 12$ and $t_{il}^{\text{from}} = 8$ to $t_{il}^{\text{to}} = 18$. The degree is depicted in blue and represents the number of neighbors of *i* at the time *t*. The kinetic degree, depicted in red, represents the sum of

the probabilities of all the links between i and its neighbors. We notice that the kinetic degree decreases faster than the degree, therefore, for every time t, the degree of i is superior to its kinetic degree. This characteristic will have consequences on the threshold function for the KDTM algorithm.

With this degree, we can predict the number of neighbors of each node at a time t with a smaller number of beacons being broadcasted. To complete this prediction, [21] also adds rules on when to send beacon messages.

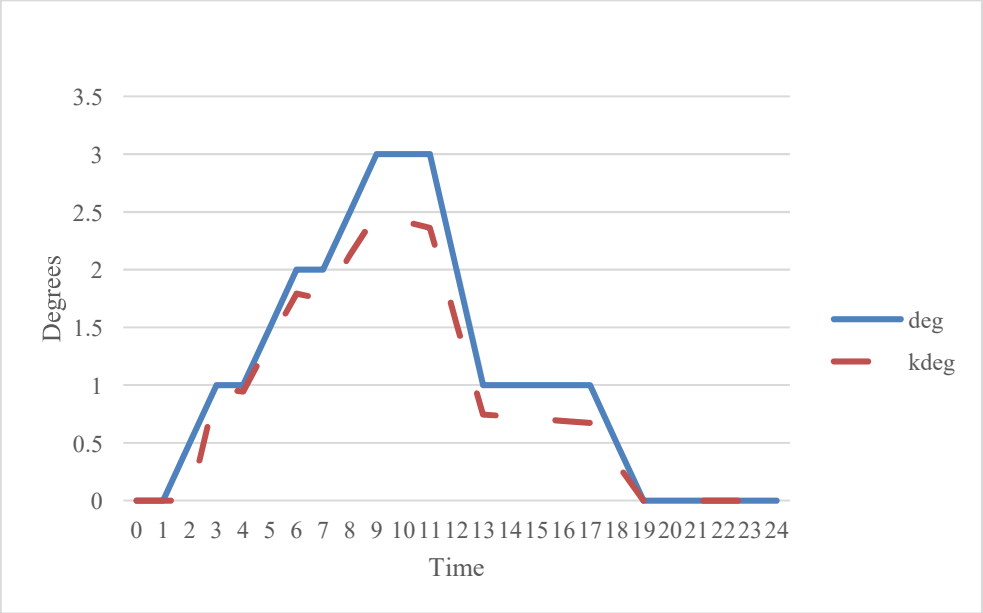


Figure 10: Node degree and kinetic degree.

3.3 Beacon broadcasting rules

Three rules are introduced on when to send hello messages and on how long to keep receiving information so that the kinetic degree prediction is relevant:

- **Constant Degree Detection:** Each node tries to maintain a constant number of neighbors. Therefore, when a node leaves a neighborhood, it will start sending beacon messages. Figure 11 presents the node i losing

its neighbor *k.* Therefore, it will directly advertise its position thanks to the constant degree detection algorithm. Then, the node *j* will receive the node *i*'s position and velocity.

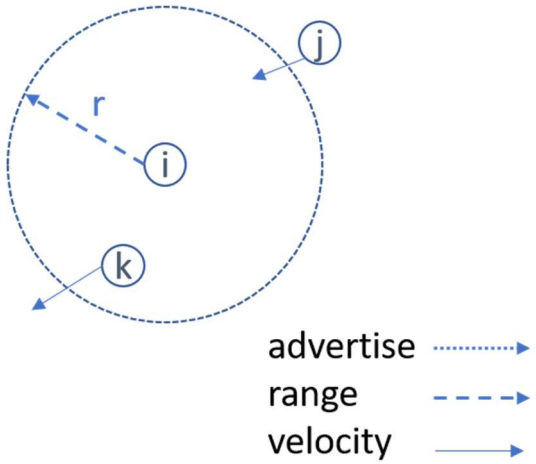


Figure 11: Constant degree detection.

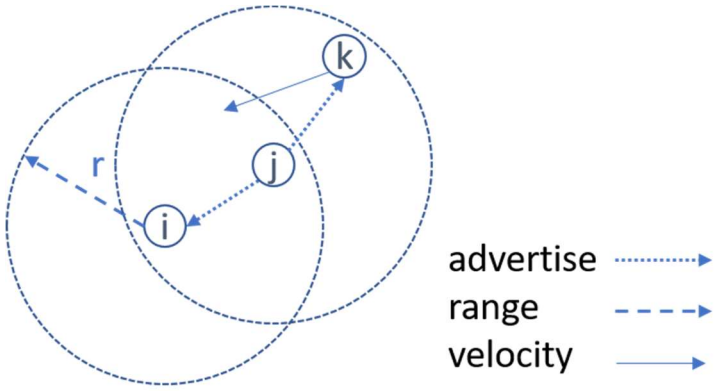


Figure 12: Implicit Detection.

- **Implicit Detection:** When a node *j* enters the transmission range of a node *i*, *i* will send *j* information about *i*'s neighbors that cannot yet be seen by *j* but that are predicted to reach *j*'s transmission range. Figure 12 represents the implicit detection algorithm. Indeed, when the node *k* enters the node

j's neighborhood, *j* will advertise the node *i*'s position and velocity to the node *k*, as well as the node *k* position and velocity to the node *i*.

- **Adaptive Coverage Detection:** A node will send a beacon when it has moved from a distance inferior or equal to its transmission range. Figure 13 represents the adaptive coverage detection, for a coefficient *n*. When the node *i* has moved of a distance of *n*r* (with *r* being *i*'s transmission range), it will advertise its velocity and position to the nodes around it.

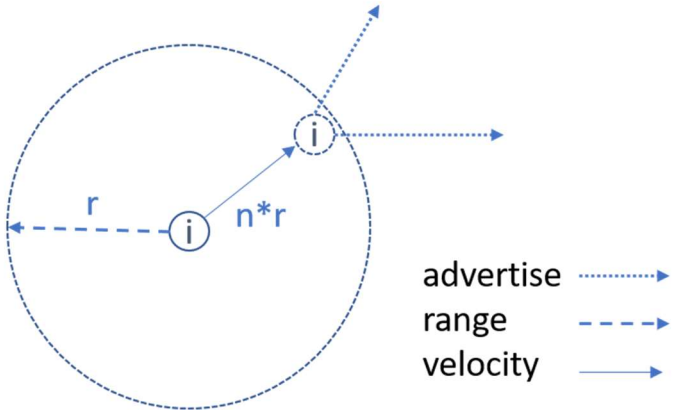


Figure 13: Adaptive Coverage Detection.

By combining the Kinetic Degree to these three rules, we should be able to decrease the number of beacons sent by the DTM protocol. But we want to study the impact on the reachability, knowing that the distance to mean threshold is based on the number of neighbors of a node.

3.4 Proposed Kinetic Distance-To-Mean Protocol (KDTM)

This proposed algorithm improves the heuristic method of the distance to mean algorithm introduced in [51] by using the non-periodic beaconing model developed in [21]. Therefore, the DTM threshold function of Eqn. (1) will be modified: the number of

neighbors in DTM is defined by an integer but in KDTM, it will be defined as a degree function of time, like in Eqn. (12). The threshold function for the node i is then defined by Eqn. (13) as follow:

$$Mc(t) = A - B\, e^{-C \cdot \widehat{Deg_i}(t)} \tag{13}$$

Where:

- *A, B, C* are constant parameters defined experimentally with a brute force to get the best results.
- $Deg_i(t)$ is defined by Eqn. (12)

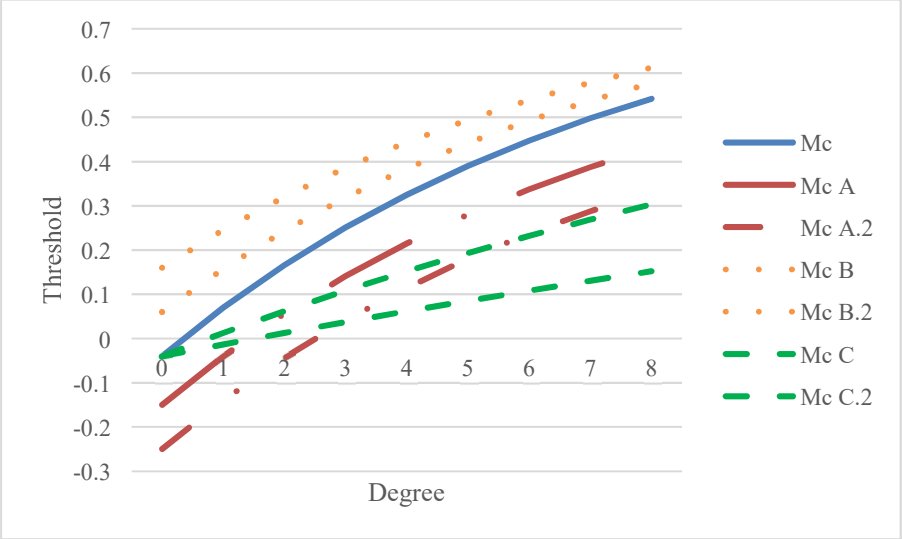


Figure 14: Variation of threshold function with the parameters.

Figure 14 presents the variation of the threshold function depending on the values chosen for *A*, *B* and C. *Mc A* and *Mc A.2* present its evolution for a decreasing value of the *A* constant. *Mc B* and *Mc C* represents a decrease of the constants *B* and *C* respectively. By decreasing or increasing these coefficients, we influence the threshold value for a certain degree. Therefore, we impact directly the number of retransmissions. We do not

obtain the same results between DTM and KDTM for the same threshold function and scenario, A, B and C must be selected independently.

3.5 Conclusion

In this chapter, we described the DTM algorithm logic and detailed all the steps it goes through. We explained that the gathering of neighbors' information requires periodical beacon message sending. In order to decrease the overhead generated by such a procedure, we use the kinetic graph model, which aims to predict the neighbors' trajectories and defines aperiodic beacon sending rules. Finally, we proposed the KDTM algorithm, which uses the distance-to-mean as next-hop selection method and the kinetic graph prediction model to calculate the number of neighbors and to define a beacons interval. In the next chapter, we will explain the implementation of the DTM algorithm in the Network Simulator NS3, first step to implement the KDTM protocol in NS3.

Chapter 4: Distance-To-Mean (DTM) Implementation in NS3

The network simulator NS3 does not include a package containing the DTM protocol, we therefore propose its implementation in this thesis. In this section, we will firstly present the components needed to implement the protocol, then, the implementation of the algorithm operations.

4.1 Implementation packaging details

Like most protocols in NS3, DTM needs several classes to perform correctly. Indeed, we need at least three models for the DTM algorithm: the position table (dtm-ptable), the warning message queue (dtm-wqueue) and the packet header definition (dtm-packet):

•**dtm-ptable:** Used to keep track of a node's neighbors, this table is instantiated for each node and it registers its neighbors for a defined time after receiving beacons. The table also updates the DTM threshold value based on the number of neighbors. For the table format, we use a map sorted by the neighbors' id and containing the neighbors' position and velocity and the expired time of the row.

- **dtm-wqueue:** The warning message queue is used to store received warning messages waiting to be forwarded. Each message meeting the specifications to be forwarded is assigned a back off timer according to Eqn. (4). When this time is reached, the message is either deleted or forwarded. In this last case, the queue keeps in memory the message's id in order to ignore further rebroadcasting of this message. The class queue

entry is defined to store the information of the packet. A map of queue entries indexed on their message id is used as data structure for the queue. To calculate the distance to mean properly, we register each received message, even if the message's id has already been received in the past.

- **dtm-packet:** The dtm packet header defines two header types, the DTM_HELLO and DTM_WARNING, which define hello message headers and warning message headers respectively. The hello message header aims to advertise the sender position and velocity; therefore, it contains the following fields: sender's id, sender's position (in Cartesian coordinates) and sender's velocity (in Cartesian coordinates). The warning message header is used to encapsulate the data. It needs to contain all the information useful to the routing. It thus contains the following attributes: the source's id, the previous hop's id, the message's id, the previous hop's position (in Cartesian coordinates). Finally, headers in NS3 inherit the NS3 class header. They thus need to implement the abstract function of serialization and deserialization required by network transmissions.

Now that the packaging details have been detailed, we will talk more about the implementation of the protocol and its algorithms.

4.2 Protocols details and implementations

The implementation of the protocol can be explained by different phases connected to their roles. We will list those as follows: the hello message generation, the warning message generation, the receive packet function and the forward packet function.

- **Hello message generation:** The generation of a Hello message consists of several steps. The first step is to generate a type header of DTM_HELLO, indicating the type of message sent. The second step is to obtain the node's identifier, velocity and position and to store them in a hello message header. The third step is to encapsulate the packet in the hello message header and then in the type header. Finally, we open a socket and broadcast the new generated hello message over the network. This task is scheduled to a chosen beacon interval.

  • **Warning message generation:** The warning message generation is follows several steps. The first one is to generate a type header of DTM_WARNING type. The second is to obtain the node's identifier, velocity and position and to use those values to create a warning message header. The third one consists in encapsulating the data packet first in the warning header, then in the type header. Finally, we broadcast the warning message over the network.

- **Receive packet function:** The first step of the receive function is to recognize the type of the message received by peeking at its type header. Then, depending on the type of message (DTM_HELLO or DTM_WARNING), the algorithm changes.
  - DTM_HELLO: if the received message is a beacon, the first step is to peek from the header the sender's identifier, position and velocity. Then, if the sender is not already known as a neighbor, we add an

entry to the neighbor table. Otherwise, we update the timer of the existing entry.

- DTM_WARNING: if the received message is a warning message, the first step is to determine if the message has already been sent, forwarded or ignored by the receiving node. If not, we calculate a back off timer based on Eqn. (4) and the information contained in the warning header. Then, we create a queue entry and insert it to the list in the queue at the index of the message's id. Then, we schedule the forward function at the back off time.

- **Forward packet function:** The forward function is only invoked from the receive function for a specified message to forward, when its back off timer is over. Therefore, the first step, according to the DTM algorithm, is to test whether the message is the last one that was received with this message id. If not, we exit the forward function and the decision to forward is reported to the new defined back off timer. However, if the entry is the last one for this message id, we calculate the spatial distribution of the senders of this message and the receiver's distance-to-mean using Eqns. (2) and (3). Then we compare the obtained value to the threshold stored in the neighbors table of the node. If the distance-to-mean is lower than the threshold, the message is not retransmitted and classified as ignored in the queue. If the distance-to-mean is higher than the threshold, we update the packet's warning header with the receiver node's information and forward the message. The message is then classified as forwarded in the queue.

This proposed implementation of DTM respects the steps of the algorithm and is fully functional in the network simulator NS3. The implementation of KDTM is based on this first implementation, however, several steps and data structures are updated to fit the characteristics of the new algorithm.

4.3 Conclusion

In this thesis, we implement the DTM protocol as a base for the KDTM algorithm and to run simulations with it. The implementation of DTM requires several components such as a neighbor table, a network queue and an implementation of the protocol logic. Thereby, we implement the beacon sending processes, the receive algorithm, the warning algorithm and finally, the forwarding algorithm. The implemented DTM algorithm is used as a base to implement the KDTM algorithm as explained in the next chapter.

Chapter 5: Kinetic Distance-To-Mean (KDTM) Implementation in NS3

In this chapter, we will detail the implementation of the KDTM algorithm proposed in this thesis in the network simulator NS3. This algorithm being based on the DTM algorithm, some components are similar. This is why this chapter will reference the previous one on data structures and functions that are the same and list the differences. Firstly, this chapter details the class models needed in KDTM and secondly, the implementation of the algorithm in detail.

5.1 Implementation packaging details

The KDTM algorithm requires the same classes than the DTM algorithm, which are a table to store neighbors' information (kdtm-ptable), a queue for warning messages waiting to be forwarded (kdtm-wqueue) and finally, packet formats for headers definition (kdtm-packet). However, some of these present some new characteristics.

- **kdtm-ptable:** Used to keep track of node's neighbors, this table is instantiated on every node and it registers its neighbors after receiving beacons. From beacon's information and using Eqns. (5) and (6), the table calculate the times *t_from* and *t_to* that describe the time when the sender enters the neighborhood and when it exits the neighborhood respectively. *t_to* is chosen as the entry life time: when reached, the entry is deleted. The table also includes a function to obtain the KDTM Threshold value function of the time *t*. This function consists in calculating the probability of presence for each entry of the table using Eqn. (8), then in calculating the

kinetic degree of the node using Eqns. (11) and (12). Based on this degree, this function returns the threshold value of the node using Eqn. (13).

- **kdtm-wqueue:** The KDTM queue has the same role and implementation as the DTM queue.
- **kdtm-packet:** The kdtm packet header defines two header types, the KDTM_HELLO and the KDTM_WARNING, which define hello messages headers and warning messages headers respectively. The KDTM warning header is the same as the DTM warning header (detailed in the previous chapter). The DTM hello header differs: in KDTM, each node advertises more information to calculate the probability of link presence defined by Equation (8). The two new items advertised are the time when the sender's current trajectory began and its Beta coefficient, which is the inverse of its average time of maintained trajectory.

Now that the classes' format used for KDTM has been detailed, we will detail how these classes are used to implement the proposed algorithm.

5.2 Protocols details and implementations

Like in the previous chapter, we will detail the algorithm according to the different functions composing it: the hello message generation, the warning message generation, the receive packet function and the forward packet function. To work correctly, the KDTM algorithm makes nodes keep track of the begin time of their trajectory and of their average time of maintaining a constant trajectory (called Poisson coefficient in [21]).

- **Hello message generation:** The generation of Hello messages follows nearly the same steps as described in the DTM algorithm. However, in this

algorithm, the hello message also transmits the time when the node's trajectory has begun and its Poisson coefficient. Furthermore, the generation of a hello message is not periodical anymore. We implement the adaptive coverage detection rule and the constant degree detection method:

- **adaptive coverage detection:** when a vehicle moves from a defined percent *n* of its transmission range, it broadcasts a hello message. This method consists in checking the position of the node every 10 seconds and in calculating the distance traveled between these intervals. The defined coefficient *n* is fixed and directly impacts the number of hello messages sent.
- **constant degree detection:** when a vehicle's kinetic degree decreases, this vehicle broadcasts a hello message. The kinetic degree is checked every 25 seconds.

We decide not to implement the implicit detection rule from the kinetic graph model, in order to decrease as much as possible, the number of hello packets sent.

- **Warning message generation:** The warning message header being the same as the one used in DTM protocol, the warning message generation is identic to the one used by the DTM protocol.
- **Receive packet function:** Like in the DTM algorithm, the first step of the receive function is to recognize the type of received message by peeking at the type header of the packet. Then, depending on the type of message (KDTM_HELLO or KDTM_WARNING), the algorithm proceeds differently.

- KDTM_HELLO: if the received message is a beacon, the first step is to peek from the header at the sender's identifier, position, velocity, its trajectory beginning time and its Poisson coefficient. Then if the sender is not already known as a neighbor, we add an entry to the node's neighbor table containing those values. Otherwise, we update the timer of the existing entry by calculating the news t^{from} and t^{to} using Eqn (6).
- KDTM_WARNING: if the received message is a warning message, we proceed like in the DTM algorithm: the first step is to determine if the message has already been sent, forwarded or ignored by the receiving node. If not, we calculate a back off timer based on Eqn. (4) and the information contained in the warning header. Then, we create a queue entry and insert it to the queue at the index of the message's id. Finally, we schedule the forward function at the back off timer end.

- **Forward packet function:** Like in the DTM algorithm, the forward function is only invoked from the receive function for a specified message to forward when its back off timer is over. Therefore, the first step is to test whether the message is the last one received with this message id. If not, we exit the forward function and the decision to forward is reported to the newly defined back off timer.

If the entry is the last one for this message id, we calculate the spatial distribution of the senders of this message and the receiver's distance-to-mean using Eqns. (2) and

(3). Then, we calculate the threshold at time *t* and from the neighbors' table entries using Eqn. (13). We compare the threshold and the distance-to-mean obtained: if the distance-to-mean is lower than the threshold, the message is not retransmitted and it is classified as ignored in the queue. If the distance-to-mean is higher than the threshold, we update the packet's warning header with the receiver node's information and forward the message. The message is then classified as forwarded in the queue.

5.3 Conclusion

Based on the DTM algorithm implemented in the previous chapter, we implement the KDTM algorithm by modifying the neighbor table and the beacon sending method. Furthermore, we adapt the kinetic graph model to predict the number of neighbors and to choose the aperiodic method used to send beacons. The implementation of the KDTM is then ready to be used in a simulation scenario defined in the network simulator NS3. In the next chapter, we will detail the chosen scenarios and their implementation.

Chapter 6: Simulation Scenario

This chapter will explain the scenarios chosen to evaluate the new protocol KDTM. As explained in Chapter 4, a VANET simulation scenario is defined by its route topology and its network component. Therefore, we will detail these components in two distinct sections.

6.1 Route topology and Mobility Scenarios

For the evaluation of the proposed KDTM algorithm, two scenarios are considered. The first one represents an urban scenario aiming to evaluate the protocol behavior in cities or in grid topologies. The second one is a highway scenario that aims to represent the protocol behavior on a linear road.

6.1.1 Urban scenario

To simulate a vehicle's mobility in an urban scenario, we use the software Simulation of Urban MObility (SUMO). Using SUMO, we generate a three by three Manhattan Grid with a size of 1km by 1km, with 500m between each intersection for the road topology. Each edge road is a two-way road, with two lines for each way. The speed limit is of 50 kilometers per hour (about 30 miles per hour and 13 meters per second) for the whole map. The velocity model used for cars is the car following model, in which every car maintains the same speed as the leading vehicle. For each simulation, the cars' positions and paths are randomly generated using the python script "randomTrips.py" (provided by SUMO). Figure 15 presents the generated map in the SUMO-GUI visual interface for a simulation with 200 vehicles (represented by yellow car shapes on the map).

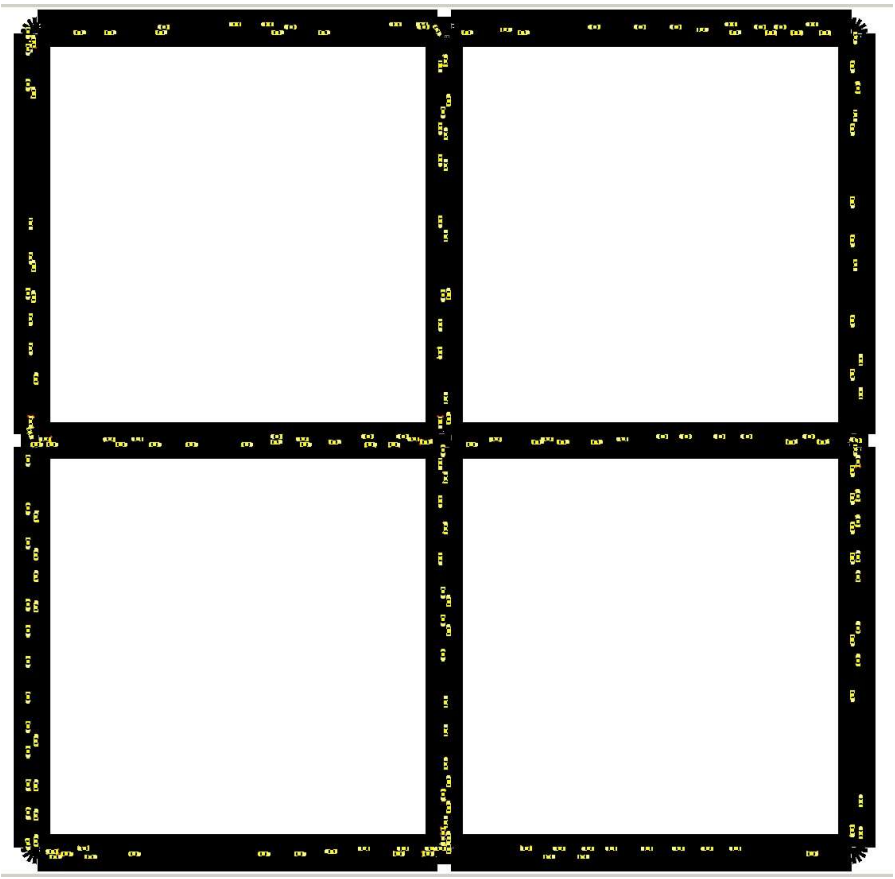


Figure 15: Urban scenario map (200 vehicles).

6.1.2 Highway scenario

The highway scenario is implemented using the simulator NS3. The mobility model used is NS3's constant speed mobility model, which gives vehicles a constant speed between each trajectory. Vehicles are allocated a constant speed, varying from 20 to 30 meters per second. Then, the vehicles are positioned using NS3's random rectangle position allocator, which randomly places vehicles on a straight line.

6.2 Network Simulation Implementation details

This section contains the implementation of a VANET scenario using NS3. Such a scenario can be divided in several operations: nodes creation, devices creation and the

implementation of the communication protocol. In this section, we will look at each of these operations.

6.2.1 Node Creation

In NS3, we define nodes by their position and mobility. In our scenarios, nodes represent vehicles. Nodes creation consists in the instantiation of a NS3 NodeContainer. Then, we associate to each node the mobility, defined by the route topology and mobility scenario. For each node, it will define a trajectory and a speed during the simulation.

6.2.2 Devices Creation and WAVE 802.11p Protocols Simulation Model

In this scenario, we decide to use NS3's WiFi80211p implemented communication protocol, which is basically the WAVE protocol without the multi-channel Quality of Service (named NQoSWaveHelper in NS3). Indeed, to evaluate the routing protocol, we do not need multi-channel communication. Therefore, the IEEE 802.11p, which is the physical layer of the WAVE protocol, is enough. To simulate the physical layer and the WiFi channel, we use two different models, based on an NS3 implementation of Yet Another Network Simulator (YANS) WiFi model, detailed in [58]. This implementation allows us to choose the transmission power, the gain for transmission and reception, the energy detection threshold, and other parameters. For our simulation we fix the range of transmission between 250 and 300 meters by modifying the transmission power. This corresponds to the range of transmissions used in the simulation in [1]. Now that each WiFi device has been created, we install on each node one device that will deal with communication for this node.

Finally, in the initialization phase, we choose the network Internet Protocol Address v4 (IPv4) and mask of the network with the broadcast address. With this information, we install on each node a socket used as a receiver sink ready to receive broadcasted messages.

6.3 Conclusion

In this chapter, we detailed the two different scenarios that will be used for the simulation. We explained how we implemented urban and highway scenario in the different simulators. Furthermore, we detailed the generation of road topology in SUMO for the urban scenario and NS3 for the highway scenario. Finally, we explained the other components of a vehicular network simulation scenario, such us the YANS model used to simulate IEEE 802.11p channel. We also used the IEEE 802.11p as a simulated physical transmission protocol. In the next chapter, we will show and interpret the results of the simulation using the parameters detailed in this chapter.

Chapter 7: Results

With our scenario in hand, we are now ready to evaluate the KDTM and the implementation of the DTM protocol. The goal is to simulate those two protocols on the two distinct scenarios defined in the previous chapter. We compare the results obtained on different performance parameters such as the reachability, the number of rebroadcast per covered nodes and the overhead.

7.1 Vehicular scenario simulation and simulation tools

In this chapter, we will define the components of a vehicular scenario. Then, we will present the two different tools used in this thesis to run simulations over a VANET. A basic vehicular scenario is built on several components: nodes and mobility, network devices, network routing and finally, application.

- **Nodes & Mobility:** This component defines vehicles and RSUs and their mobility during the simulation. In this thesis, we will use two different software to handle this component: the network simulator NS3 and the road topology and node mobility simulator SUMO.
- **Network Devices:** This component presents the devices embarked in the vehicles. It defines the protocol used by this device (WiFi, WAVE, 802.11p…). In this thesis, this component is also handled with NS3.
- **Network / Routing:** This component presents the way the nodes will handle routing in the network. This is the component where the implementation of the DTM and KDTM algorithms take place. This component is coded in NS3 in this thesis.

- **Applications:** This component represents the highest layer of network protocol stacks. In this thesis, this layer is ignored because we are only interested in the lower level, especially the routing component.

Now that we have presented the different components required for a vehicular scenario a simulation, we will present the two-simulator used in this thesis, NS3 and SUMO. The paper [59] presents a survey on all the different simulators available to create vehicular scenarios on VANET.

- **Network Simulator (NS3):** NS3 is a discrete-event network simulator developed primarily for research and educational matters. This simulator is an open source software under GNU GPLv2 license. It presents a full set of models to simulate from nodes mobility to routing algorithms in a vehicular scenario. Developed in C++, NS3 is one of the most complete network simulator in use in research over VANETs.

- **Simulation of Urban Mobility (SUMO) [60]:** Designed to handle large scaled road topology, SUMO is an open source road simulation and traffic generator. It uses xml files to define road characteristics and nodes' mobility, which makes it highly portable. SUMO also includes a graphical interface coded in OPEN GL (GUI) to visualize simulations. It provides multi-lane traffic generation, with priority lane and cross-road light simulation.

Now that we detailed the different tools used to create simulation scenarios in VANET, we will introduce the performances parameters evaluated during the different simulations ran for the evaluation of the KDTM algorithm.

7.2 Performance Parameters Evaluated

- **Reachability:** The reachability represents the number of vehicles reached by a warning message, over the total number of vehicles. This value is a critical value in broadcasting routing algorithms, because these algorithms aim to disseminate messages over the whole network. In our case, we want to control the reachability of DTM so that it is not impacted by the decreasing of number of hello packets sent, because the threshold function depends directly on the neighbor table impacted by the Kinetic Graph Model used in KDTM.
- **Rebroadcast per cover nodes:** Evaluates the number of rebroadcasts during the transmission of a warning message. This value is calculated with the ratio of the number of rebroadcasts over the number of nodes reached by the warning message. Broadcasting routing algorithms aim to decrease this value and thereby, the overall overhead. In our scenario, we measure this value to evaluate if the neighbor prediction has an impact on the number of rebroadcasts per cover nodes.
    - **Overhead:** The overhead is calculated with the ratio between the number of bytes sent and the number of nodes that received the warning message. The overhead considers the number of bytes sent with the hello message. Therefore, this is the value we want to decrease with the KDTM

algorithm by influencing the number of hello messages sent. Therefore, we also measure separately the overhead generated by the beacons and by the warning message (forwarding included).

Knowing the different parameters evaluated, we will now present the different results obtained for the two protocols DTM and KDTM over the two different scenarios, urban and highway environment.

7.3 Results

The validation of the KDTM routing protocol is realized over the two scenarios: urban and highway environment. For both scenarios, we compare the results with the DTM algorithm, simulated with a beacon interval of 15 seconds and the threshold function shown in Eqn. (14). This threshold function is also used for the KDTM algorithm, except that the number of neighbors N is replaced by the Kinetic degree of the node.

$$M(N) = 0.8 - 0.95e^{-0.06N} \tag{14}$$

7.3.1 Highway scenario

The parameters of the simulation are shown in Table 2.

Table 2 - Highway scenario parameters

highway scenario	Values
speed	20-30 m/s
portion size	1000 m
range	250-300 m
nb of nodes	20,50,100,200,400
packet size	200 Kb
simulation time	300s

For the highway simulation scenario, we run simulations over three versions of the protocol KDTM named KDTM -1, KDTM -2 and KDTM -3. The differences between those instances are the neighbor's detection method parameters. Indeed, the parameter of the adaptive coverage detection *n* impacts the number of rebroadcasts, especially when the vehicle speed is high, like in this scenario. We therefore use two different values, as shown in Table 3. We also run the simulation with and without the constant degree detection method: this allows to evaluate its influence on the number of bytes sent.

Table 3: Protocol simulated in highway scenario.

Protocol	Adaptive coverage detection n	Constant degree detection
KDTM -1	2/3	no
KDTM -2	2	no
KDTM -3	2	yes

7.3.1.1 Reachability

The reachability for a similar threshold function is compared between the DTM and the KDTM algorithm in Fig. 16. We observe that the reachability on the highway scenario is improved when the network is dense. Increasing the adaptive coverage method coefficient also increase the reachability.

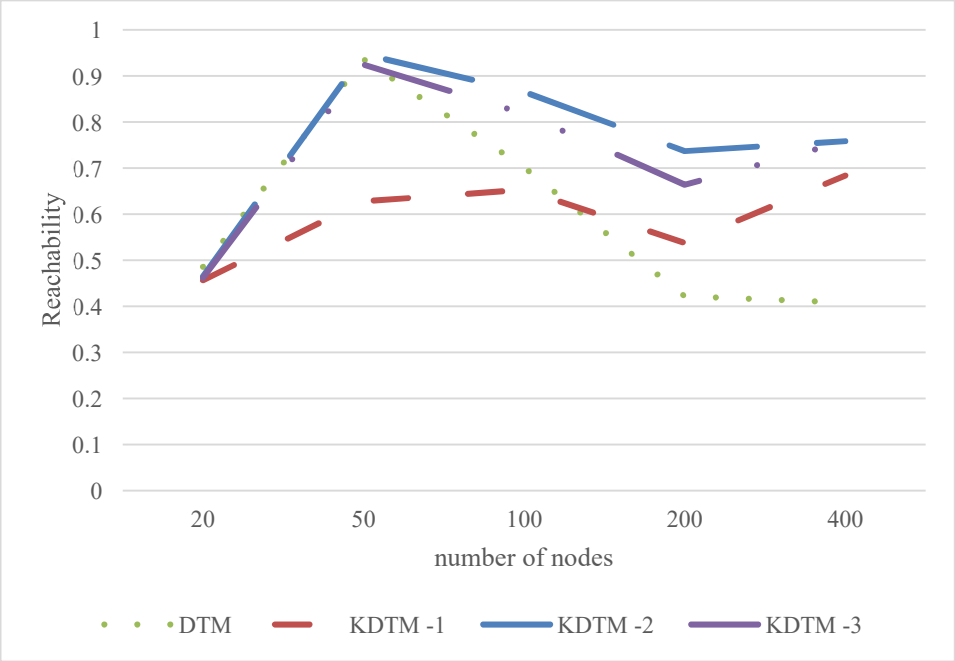


Figure 16: Reachability in highway environment.

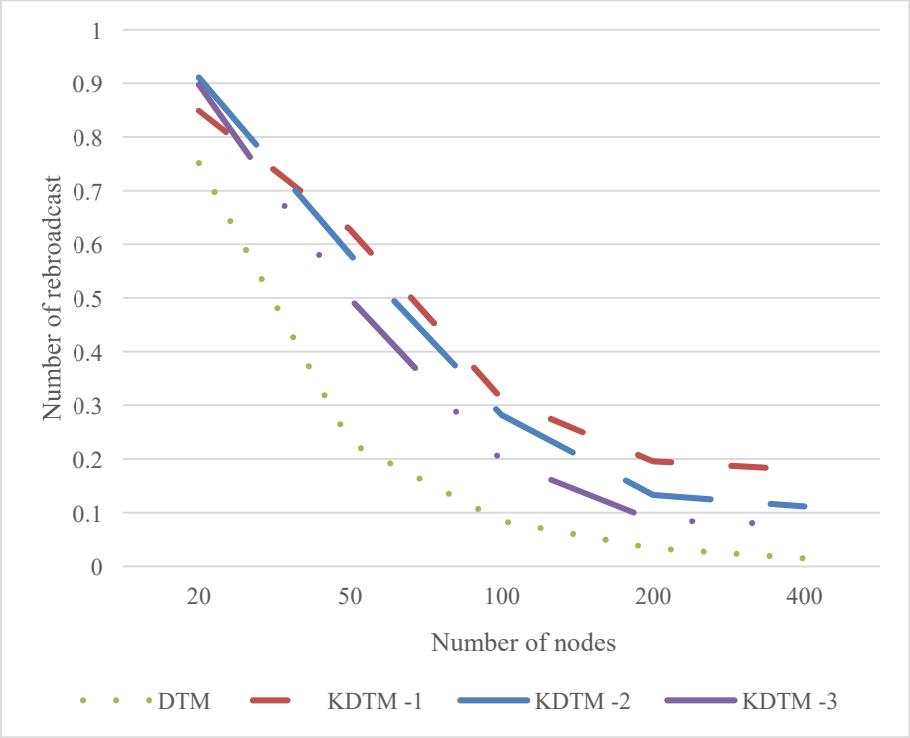


Figure 17: Number of rebroadcast per covered node in highway environment.

7.3.1.1 Rebroadcast per covered node

Implementing the constant degree detection decrease the reachability around 0.05%, however, it decreases consequently the number of rebroadcasting per covered nodes as shown in Fig. 17. On average, the KDTM algorithm increase the number of forwarding by 0.21%.

7.3.1.2 Overhead

Figure 18 depicts the overhead generated by the warning message in the highway environment. This value is directly connected to the number of rebroadcasts per covered node. Therefore, it is also connected to the threshold function. We observe that the KDTM algorithm sends in average 41% less bytes of the warning message on the network over the simulation period than DTM. KDTM -3 presents the best results, with an average of 58% less.

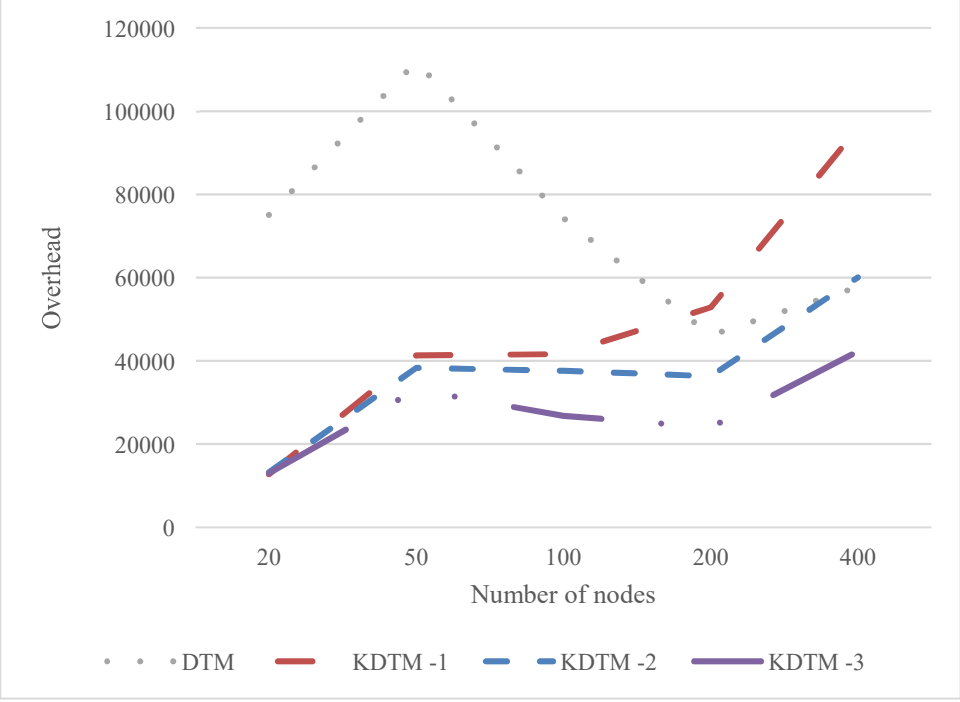


Figure 18: Overhead from warning messages in highway environment.

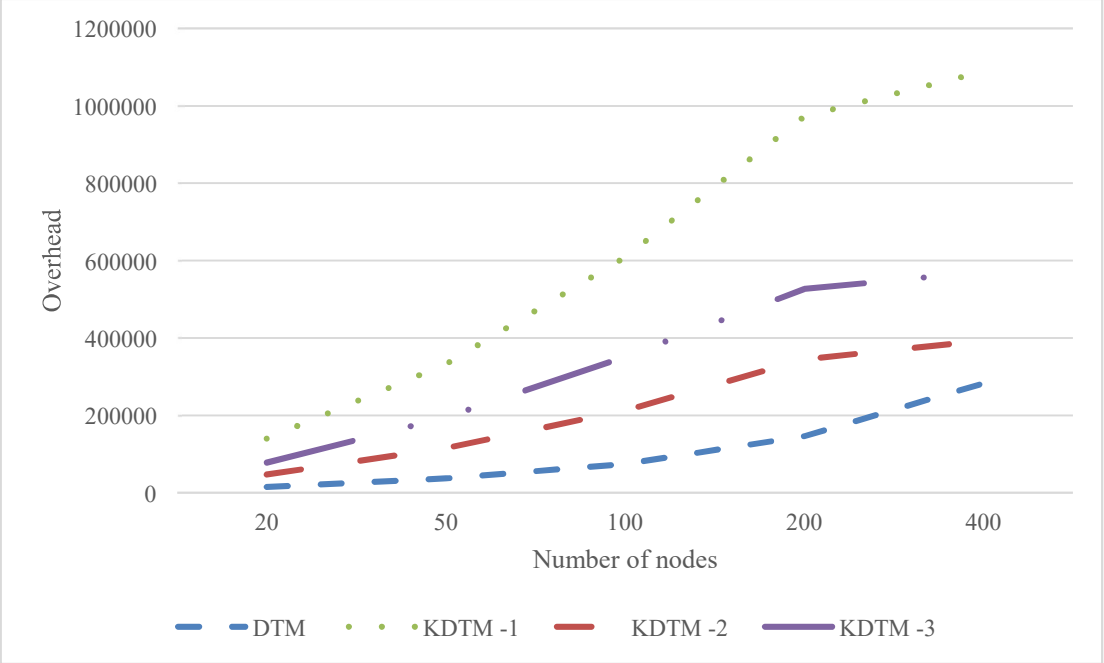


Figure 19: Overhead from hello messages in highway environment.

Figure 19 represents the overhead created by hello packets. We observe that the KDTM algorithm generates an overhead significantly more important than the DTM algorithm. However, by increasing the coefficient of the adaptive coverage detection method (for KDTM -2 and KDTM -3) the overhead generated is less important. Therefore, KDTM -3, which presents good results in terms of reachability and number of rebroadcasts, shows an increment of 200% compared to the DTM algorithm in terms of hello packet overhead.

The global overhead is calculated by adding the overhead from the beacons and from the warning message. Figure 20 represents the global overhead in the highway environment. The KDTM -3, which presents the better reachability, still presents an increase of 200% from the DTM algorithm in terms of overhead.

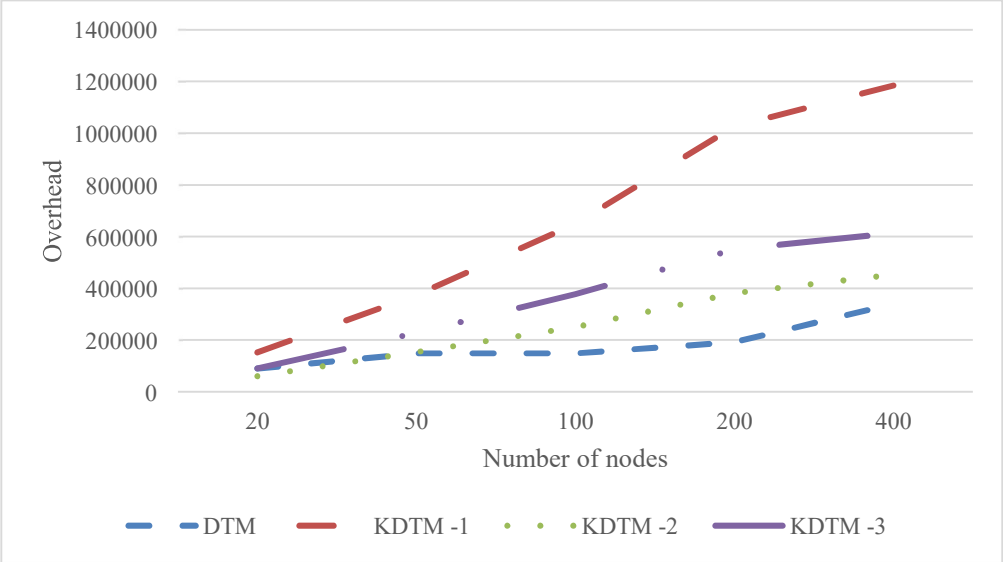


Figure 20: Global overhead in Highway environment.

The results for the highway scenario present an average progression of 15% of reachability. However, the cost in terms of bandwidth consumption (plus 200%) and number of rebroadcasts per covered nodes (plus 17%) is too high. Increasing the adaptive coverage coefficient to more than 2 and improving the protocol with a new detection rule, more suited to high speed, could be a solution to improve those values.

7.3.2 Urban scenario

The urban scenario simulation evaluates the KDTM protocol with the parameters described in Table 4. We run the simulations presented in Table 5 over the map generated with SUMO. The simulation time is 300 seconds and the warning message is sent every 5 seconds from the time t=50s to the time t=250s. The sender is chosen randomly among the moving vehicles.

Table 4: Urban scenario protocol measured.

Protocol	Adaptive coverage n	Constant degree detection
KDTM -1	2/3	no
KDTM -2	2/3	yes
KDTM -3	2	yes

Table 5: Urban Scenario parameters.

Urban scenario	Values
speed	13 m/s
Grid	3x3, 1000x1000
range	250-300 m
nb of nodes	20,50,100,200,400
packet size	200 kb
ration range	2/3
simulation time	300s

7.3.2.1 Reachability

In terms of reachability, the KDTM algorithm shows a decrement of 0.13% on average independently of parameters, as shown in Figure 21. However, this decrement is due to the threshold function, which is also used for DTM and KDTM.

The number of rebroadcasts per covered node is on average 14% lower for the KDTM algorithm than for the DTM algorithm. Figure 22 presents these results. Again, this value depends on the threshold function.

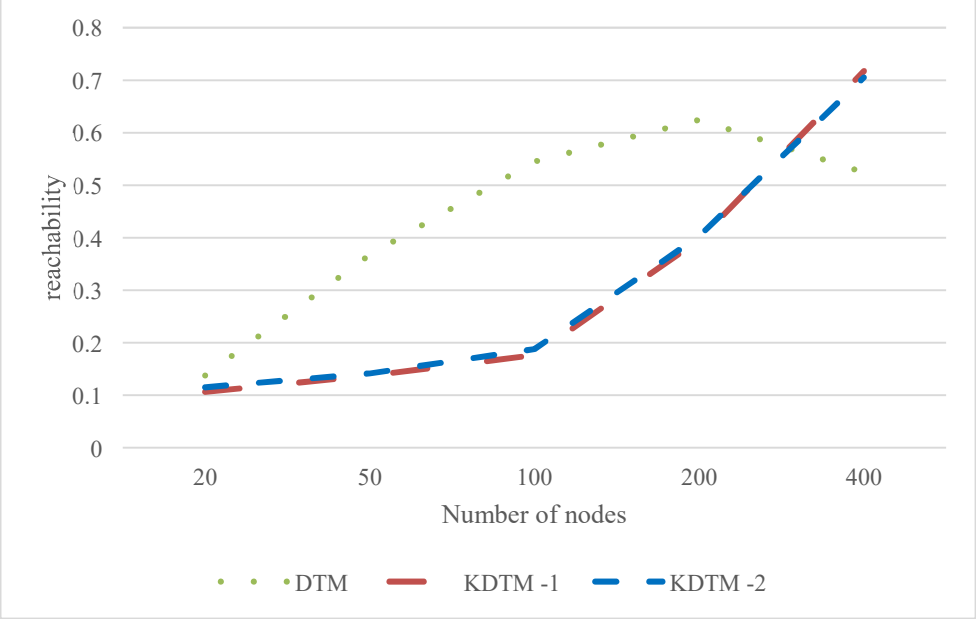


Figure 21: Reachability in Urban environment.

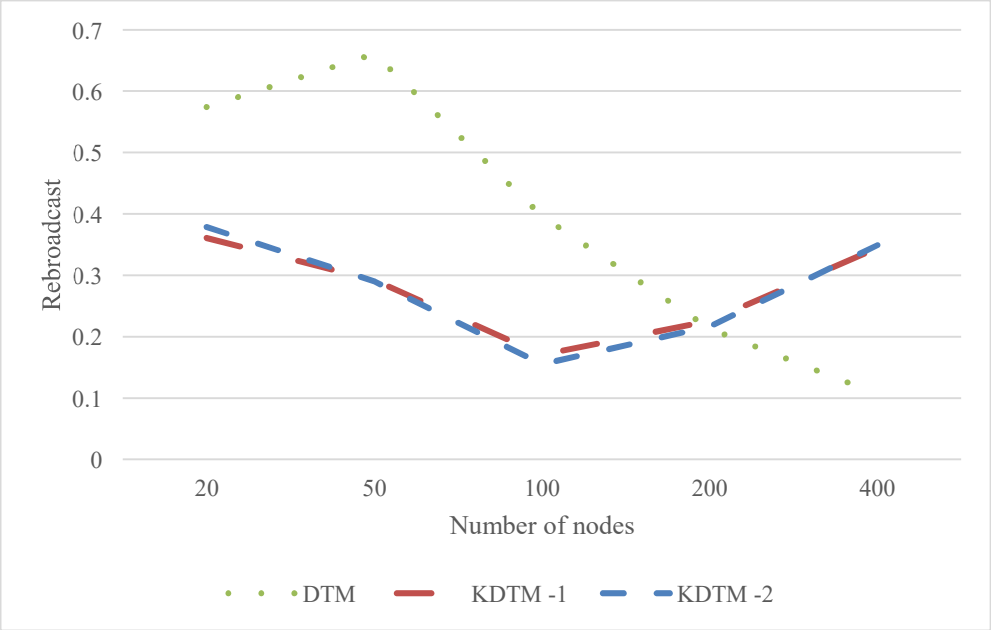


Figure 22: Number of rebroadcasting per covered node in urban environment.

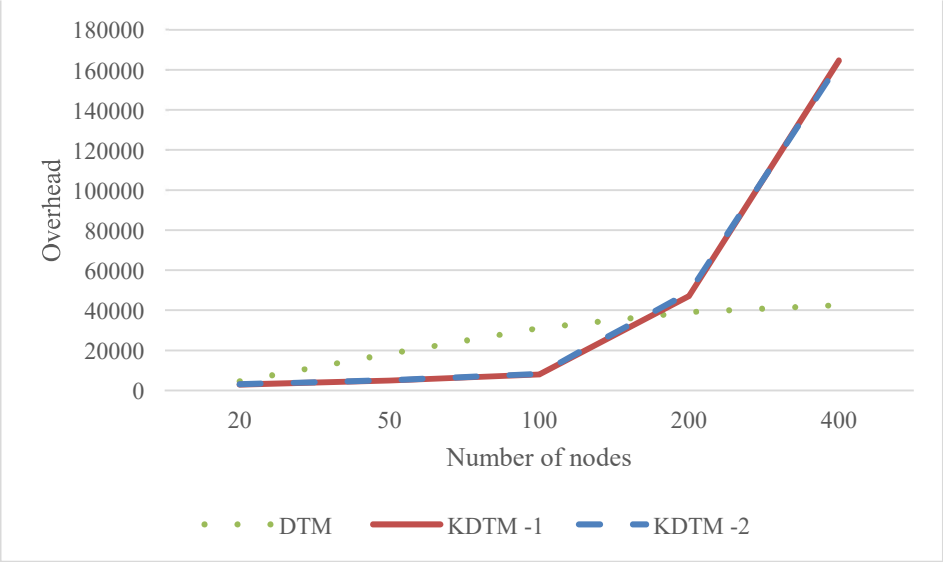


Figure 23: Overhead generated by warning messages in urban environment.

The overhead generated by warning messages is represented in Fig. 23. We observe that the KDTM algorithm presents a lower overhead in small networks but a higher one for dense networks.

The overhead generated by the beacons is represented on Fig. 24: we observe a consequent decrease of 45% on average. Those results demonstrate that the KDTM protocol in urban environment presents a lower overhead with the adaptive coverage detection and the constant degree detection (KDTM -2). Those results are validated by the representation of the global overhead in Fig. 25.

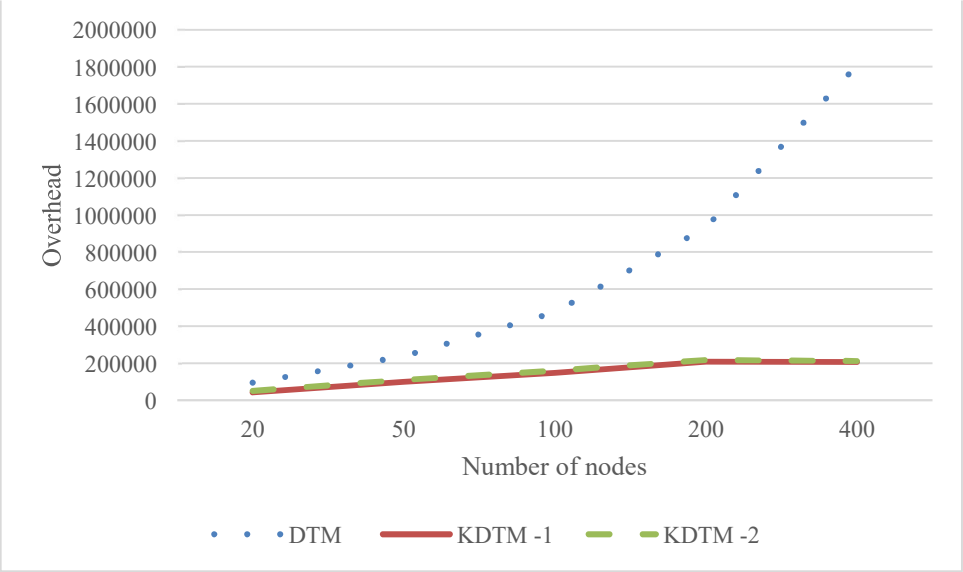


Figure 24: Overhead generated by hello messages in urban environment.

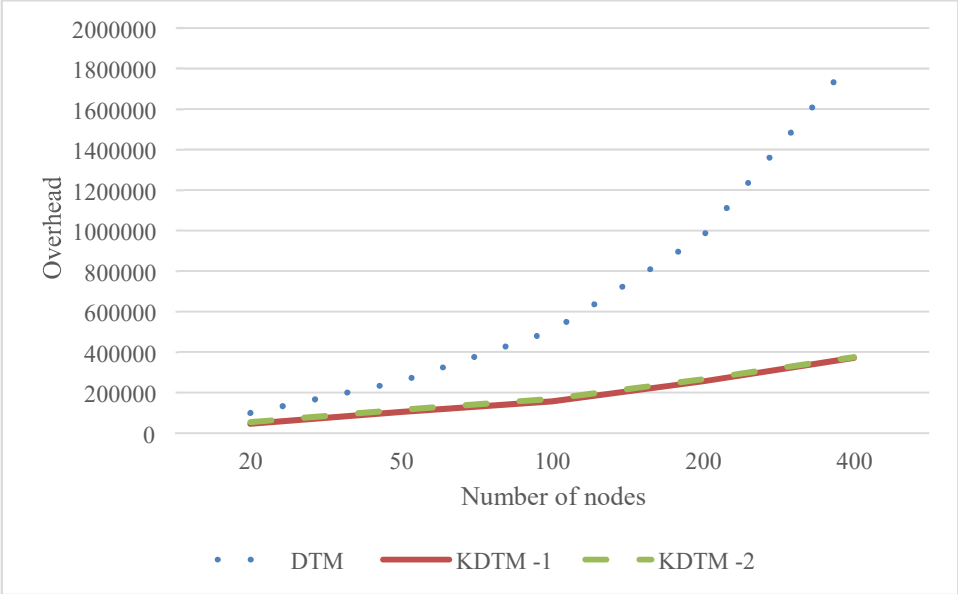


Figure 25: Global overhead in urban environment.

7.4 Conclusion

The results presented in this chapter show different characteristics depending on the scenario simulated. In the highway scenario, the high speed of the vehicles implies that the aperiodic messages are sent more often than the periodically generated messages.

By increasing the adaptive coverage detection, we therefore decrease the overhead to get closer to the DTM algorithm (like KDTM -2 and KDTM -3 in Fig. 20). However, the reachability and rebroadcast per covered node are better than the DTM in the highway environment. In the urban environment, the overhead generated by KDTM is significantly lower than the one generated by DTM (as shown in Fig. 25). However, the reachability is lower, especially for small networks (as shown in Fig. 21). We can draw two conclusions from these results:

- The speed impacts the overhead, mainly due to the coverage adaptive detection rule but also the constant degree detection rule. Therefore, a speed adaptive rule could be added to the KDTM algorithm, or the adaptive coverage detection coefficient could be defined as a function of the node velocity. Those two approaches are considered as future work.
- The Threshold value, function of the number of neighbors, used in the DTM algorithm and adapted to the KDTM algorithm is not perfectly adapted to the kinetic degree for different nodes density. Therefore, adding fuzzification methods to the definition of the threshold is considered as a future work.

However, the KDTM protocol shows interesting results in dense and slow networks in terms of overhead and reachability (Figure 21). With a more adaptive threshold function, it could considerably reduce the overhead of distance-to-mean based routing protocols.

Chapter 8: Conclusion and future work

In this thesis, we propose the Kinetic Distance-to-Mean protocol (KDTM) routing algorithm for Vehicular Ad-Hoc Networks (VANETs). This algorithm can be classified as a hybrid broadcasting geographical routing algorithm, because it uses the geographical location of vehicles and broadcasting routing techniques. The next hop selection uses the distance-to-mean (DTM) heuristic, which aims to decrease the number of rebroadcasts per covered nodes in VANETs, compared to classical distance based algorithms. KDTM also decreases the number of beacon messages sent using a Kinetic Graph Model, in order to decrease the overhead, compared to other neighbor aware protocols. In this thesis, we also implement the two-routing protocols DTM and KDTM in the network simulator NS3 for evaluation. Therefore, we compare KDTM to DTM in two different scenarios, an urban scenario and a highway scenario. The results differ depending on the scenario. In the highway scenario, the KDTM algorithm shows a higher reachability, but also a higher global overhead. In the urban scenario, it presents a lower reachability and a lower overhead. Those results suggest that the KDTM presents different results depending on the speed limit, which is one of the differences between the two scenarios (13 m/s for the urban scenario, 20-30 m/s for the highway scenario). Adapting a neighbor detection method could therefore be interesting. Furthermore, the threshold function used in the DTM algorithm, which was adapted for the KDTM algorithm, does not adapt well enough to the scalability of a VANET. Therefore, improving the threshold function using fuzzification techniques would be considered as future work. Those methods have shown good results, like in the fuzzy logic-based broadcast (FLB) and fuzzy logic-assisted broadcast (BEFLAB) protocol, both based on DTM with an improved threshold selection.

Finally, I would format the KDTM implementation to the NS3 standards to make it available in the NS3 library.

References

[1] M. Slavik, I. Mahgoub, and M. M. Alwakeel, "Analysis and evaluation of distance-to-mean broadcast method for VANET," *Journal of King Saud University-Computer and Information Sciences,* vol. 26, no. 1, pp. 153-160, 2014.

[2] H. Houda and M. Salah, "A survey of trajectory based data forwarding schemes for vehicular ad-hoc networks," in *IEEE International Conference on Communication Software and Networks (ICCSN),* pp. 399-404, 2015.

[3] *K R Jothi, Jeyakumar A Ebenezer, "*A Survey on Broadcasting Protocols in VANETs", *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, *2013.*

[4] C. Lemmon, S. M. Lui, and I. Lee, "Geographic forwarding and routing for adhoc wireless network: A survey," in *IEEE Fifth International Joint Conference on INC, IMS and IDC. NCM'09*, pp. 188-195, 2009.

[5] L. Kristiana, C. Schmitt, and B. Stiller, "Survey of angle-based forwarding methods in VANET communications," in *IEEE Wireless Days Conference(WD),* pp. 1-3, 2016.

[6] M. Wang, Y. Zhang, C. Li, X. Wang, and L. Zhu, "A survey on intersectionbased routing protocols in city scenario of VANETs," in *IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, pp. 821-826, 2014.

[7] S. Bitam, A. Mellouk, and S. Zeadally, "Bio-inspired routing algorithms survey for vehicular ad hoc networks," in *IEEE Communications Surveys & Tutorials,* vol. 17, no. 2, pp. 843-867, 2015.

[8] S. A. Ahmed, S. H. Ariffin, N. Fisal, S. Syed-Yusof, and N. Latif, "Survey on broadcasting in VANET," *Research Journal of Applied Sciences, Engineering and Technology,* vol. 7, no. 18, pp. 3733-3739, 2014.

[9] J. Kakarla, S. S. Sathya, and B. G. Laxmi, "A survey on routing protocols and its issues in VANET," *Int. J. Comput. Appl. Conf., ISSN, 28 (4), 0975-8887,* 2011.

[10] Z. Guoqing, M. Dejun, X. Zhong, Y. Weili, and C. Xiaoyan, "A survey on the routing schemes of urban vehicular ad hoc networks," in *27th Chinese IEEE Control Conference, CCC*, pp. 338-343, 2008.

[11] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc on-demand distance vector (AODV) routing," *Proceeding of the Second IEEE Workshop on Mobile Computer Systems and Application,* p90, 1999.

[12] H. Somnuk and M. Lerwatechakul, "Multi-hop AODV-2T," in *International Symposium on Intelligent Ubiquitous Computing and Education*, 2009.

[13] A. B. Souza, J. Celestino, F. A. Xavier, F. D. Oliveira, A. Patel, and M. Latifi, "Stable multicast trees based on Ant Colony optimization for vehicular Ad Hoc networks," in *IEEE International Conference on Information Networking (ICOIN)*, pp. 101-106, 2013.

[14] T. Clausen and P. Jacquet, "Optimized link state routing protocol (OLSR)," in *IEEE INMIC Conf.*, pp. 2070-1721, 2001.

[15] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers," in *ACM SIGCOMM computer communication review online articles*, vol. 24, no. 4, pp. 234-244, 1994.

[16] D. B. Johnson, D. A. Maltz, and J. Broch, "DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks," *Ad hoc networking,* vol. 5, pp. 139-172, 2001.

[17] Q. Liu, H. Wang, J. Kuang, Z. Wang, and Z. Bi, "WSNp1-1: M-TORA: a TORA-based multi-path routing algorithm for mobile ad hoc networks," in *IEEE Global Telecommunications Conference. GLOBECOM'06*, pp. 1-5, 2006.

[18] J. Ding, "Simulation and evaluation of the performance of fsr routing protocols based on group mobility model in mobile ad hoc," in *IEEE International Conference on Computational Intelligence and Software Engineering (CiSE)*, pp. 1-4, 2010.

[19] J. Mungara, "A unified approach to enhance the performance of zrp for manets on an urban terrain," in *IEEE International Conference on Progress in Informatics and Computing (PIC)*, vol. 1, pp. 532-536, 2010.

[20] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29-41, 1996.

[21] J. Härri, C. Bonnet, and F. Filali, "Kinetic mobility management applied to vehicular ad hoc network protocols," *Computer Communications*, vol. 31, no. 12, pp. 2907-2924, 2008.

[22] S. Bitam and A. Mellouk, "QoS swarm bee routing protocol for vehicular ad hoc networks," in *IEEE International Conference on Communications (ICC)*, pp. 1-5, 2011.

[23] S. Bitam, A. Mellouk, and S. Zeadally, "HyBR: A hybrid bio-inspired bee swarm routing protocol for safety applications in vehicular ad hoc networks (VANETs)," *Journal of Systems Architecture,* vol. 59, no. 10, pp. 953-967, 2013.

[24] U. Lee, E. Magistretti, M. Gerla, P. Bellavista, P. Lió, and K.-W. Lee, "Bioinspired multi-agent data harvesting in a proactive urban monitoring environment," *Ad Hoc Networks*, vol. 7, no. 4, pp. 725-741, 2009.

[25] Z. Mo, H. Zhu, K. Makki, and N. Pissinou, "MURU: A multi-hop routing protocol for urban vehicular ad hoc networks," in *Third Annual IEEE International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pp. 1-8, 2006.

[26] R. S. Raw and S. Das, "Performance comparison of Position based routing Protocols in vehicle-to-vehicle (V2V) Communication," *International Journal of Engineering Science and Technology*, vol. 3, no. 1, pp. 435-444, 2011.

[27] L. K. Qabajeh, M. M. Kiah, and M. Qabajeh, "A qualitative comparison of position-based routing protocols for ad-hoc networks," *International Journal of Computer Science and Network*, vol. 9, no. 2, pp. 131-140, 2009.

[28] M. Kihl, M. Sichitiu, T. Ekeroth, and M. Rozenberg, "Reliable geographical multicast routing in vehicular ad-hoc networks," in *WWIC Conf.*, pp. 315-325, 2007.

[29] J. Toutouh and E. Alba, "Parallel swarm intelligence for VANETs optimization," in *Seventh IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pp. 285-290, 2012.

[30] T. Camp, J. Boleng, and L. Wilcox, "Location information services in mobile ad hoc networks," in *IEEE International Conference on Communications (ICC)*, vol. 5, pp. 3318-3324, 2002.

[31] V. Naumov and T. R. Gross, "Connectivity-aware routing (CAR) in vehicular ad-hoc networks," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communication*, pp. 1919-1927, 2007.

[32] T. Song, W. Xia, T. Song, and L. Shen, "A cluster-based directional routing protocol in VANET," in *12th IEEE International Conference on Communication Technology (ICCT)*, pp. 1172-1175, 2010.

[33] R. R. Sahoo, R. Panda, D. K. Behera, and M. K. Naskar, "A trust based clustering with Ant Colony Routing in VANET," in *Third IEEE International Conference on Computing Communication & Networking Technologies (ICCCNT)*, pp. 1-8, 2012.

[34] H. Saleet, R. Langar, O. Basir, and R. Boutaba, "Adaptive message routing with QoS support in vehicular Ad Hoc networks," in *IEEE Global Telecommunications Conference. GLOBECOM*, pp. 1-6, 2009.

[35] H. Saleet, R. Langar, K. Naik, R. Boutaba, A. Nayak, and N. Goel, "Intersection-based geographical routing protocol for VANETs: a proposal and analysis," *IEEE Transactions on Vehicular Technology,* vol. 60, no. 9, pp. 45604574, 2011.

[36] J. Jeong, S. Guo, Y. Gu, T. He, and D. H. Du, "Trajectory-based data forwarding for light-traffic vehicular ad hoc networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, pp. 743-757, 2011.

[37] J. Jeong, S. Guo, Y. Gu, T. He, and D. H. Du, "Trajectory-based statistical forwarding for multihop infrastructure-to-vehicle data delivery," *IEEE Transactions on Mobile Computing*, vol. 11, no. 10, pp. 1523-1537, 2012.

[38] J. P. Jeong, T. He, and D. H. Du, "TMA: Trajectory-based Multi-Anycast forwarding for efficient multicast data delivery in vehicular networks," *Computer Networks*, vol. 57, no. 13, pp. 2549-2563, 2013.

[39] F. Xu, S. Guo, J. Jeong, Y. Gu, Q. Cao *et al.*, "Utilizing shared vehicle trajectories for data forwarding in vehicular networks," in *IEEE Conference INFOCOM*, pp. 441-445, 2011.

[40] P.-C. Cheng, J.-T. Weng, L.-C. Tung, K. C. Lee, M. Gerla, and J. Haerri, "GeoDTN+ Nav: a hybrid geographic and DTN routing with navigation assistance in urban vehicular networks," *MobiQuitous/ISVCS Conf.*, vol. 47, 2008.

[41] O. K. Tonguz, N. Wisitpongphan, and F. Bai, "DV-CAST: A distributed vehicular broadcast protocol for vehicular ad hoc networks," *IEEE Wireless Communications,* vol. 17, no. 2, 2010.

[42] Y. Pekşen and T. Acarman, "Relay of multi-hop safety message based on beaconing in VANET," in *IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pp. 432-436, 2012.

[43] C. Lochert, M. Mauve, H. Füßler, and H. Hartenstein, "Geographic routing in city scenarios," *ACM SIGMOBILE mobile computing and communications review*, vol. 9, no. 1, pp. 69-72, 2005.

[44] Q. Lin, C. Li, X. Wang, and L. Zhu, "A three-dimensional scenario oriented routing protocol in vehicular ad hoc networks," in *IEEE Vehicular Technology Conference (VTC Spring), 77th*, pp. 1-5, 2013.

[45] H. ALshaer and E. Horlait, "An optimized adaptive broadcast scheme for intervehicle communication," in *IEEEVehicular Technology Conference, VTC. 61st*, vol. 5, pp. 2840-2844, 2005.

[46] J.-J. Chang, Y.-H. Li, W. Liao, and C. Chang, "Intersection-based routing for urban vehicular communications with traffic-light considerations," *IEEE Wireless Communications*, vol. 19, no. 1, 2012.

[47] L. Hogie, P. Bouvry, M. Seredynski, and F. Guinand, "A bandwidth-efficient broadcasting protocol for mobile multi-hop ad hoc networks," in *IEEE International Conference on Networking, IEEE International Conference on Systems and IEEE International Conference on Mobile Communications and Learning Technologies. ICN/ICONS/MCL*, pp. 71-71, 2006.

[48] G. Danoy, B. Dorronsoro, P. Bouvry, B. Reljic, and F. Zimmer, "Multi-objective Optimization for Information Sharing in Vehicular Ad Hoc Networks," in *Springer Journal IAIT*, pp. 58-70, 2009.

[49] J. J. Durillo, A. J. Nebro, and E. Alba, "The jMetal framework for multiobjective optimization: Design and architecture," in *IEEE Congress on Evolutionary Computation (CEC)*, pp. 1-8, 2010.

[50] Y. Sung and M. Lee, "Light-weight reliable broadcast message delivery for vehicular ad-hoc networks," in *IEEE Vehicular Technology 75th Conference (VTC Spring)*, pp. 1-6, 2012.

[51] M. Slavik, I. Mahgoub, and F. N. Sibai, "The distance-to-mean broadcast method for vehicular wireless communication systems," in *IEEE International Conference on Innovations in Information Technology (IIT)*, pp. 371-374, 2011.

[52] M. Slavik and I. Mahgoub, "Spatial distribution and channel quality adaptive protocol for multihop wireless broadcast routing in VANET," *IEEE Transactions on Mobile Computing*, vol. 12, no. 4, pp. 722-734, 2013.

[53] E. Limouchi and I. Mahgoub, "Cross-layer statistical broadcast protocol with density-adaptive contention window for vehicular ad hoc networks," *Computer and Electrical Engineering and Computer Science Department, Florida Atlantic University,* 2016.

[54] M. Slavik and I. Mahgoub, "Applying machine learning to the design of multihop broadcast protocols for VANET," in *7th IEEE International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 1742-1747, 2011.

[55] E. Limouchi, I. Mahgoub, and A. Alwakeel, "Fuzzy logic-based broadcast in vehicular ad hoc networks," in *IEEE Vehicular Technology 84th Conference (VTCFall)*, pp. 1-5, 2016.

[56] E. Limouchi and I. Mahgoub, "BEFLAB: Bandwidth efficient fuzzy logicassisted broadcast for VANET," in *IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1-8, 2016.

[57] E. Limouchi and I. Mahgoub, "Intelligent hybrid adaptive broadcast for VANET," in *IEEE Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 1-7, 2016.

[58] M. Lacage and T. R. Henderson, "Yet another network simulator," in *Proceeding from the workshop on ns-2: the IP network simulator*, p. 12: ACM, 2006.

[59] F. J. Martinez, C. K. Toh, J. C. Cano, C. T. Calafate, and P. Manzoni, "A survey and comparative study of simulators for vehicular ad hoc networks (VANETs)," *Wireless Communications and Mobile Computing*, vol. 11, no. 7, pp. 813-828, 2011.

[60] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner, "SUMO (Simulation of Urban MObility)-an open-source traffic simulation," in *Proceedings of the 4th middle East Symposium on Simulation and Modelling (MESM20002)*, pp. 183-187, 2002.

Appendix A: SUMO urban map generation

- nod.xml

```
<nodes>
    <node id="00" x="0" y="0" />
    <node id="01" x="500" y="0" />
    <node id="02" x="1000" y="0" />
    <node id="10" x="0" y="500" />
    <node id="11" x="500" y="500" />
    <node id="12" x="1000" y="500" />
<node id="20" x="0" y="1000" />
    <node id="21" x="500" y="1000" />
    <node id="22" x="1000" y="1000" />
</nodes>
```

- edg.xml

```
<edges>
    <!-- Horizontal edges -->
        <!-- First row -->
            <edge id="0001" from="00" to="01" numLanes="2"
speed="13" />            <edge id="0100" from="01" to="00"
numLanes="2" speed="13" />
            <edge id="0102" from="01" to="02" numLanes="2"
speed="13" />
            <edge id="0201" from="02" to="01" numLanes="2"
speed="13" />
        <!-- Second row -->
            <edge id="1011" from="10" to="11" numLanes="2"
speed="13" />            <edge id="1110" from="11" to="10"
numLanes="2" speed="13" />
            <edge id="1112" from="11" to="12" numLanes="2"
speed="13" />            <edge id="1211" from="12" to="11"
numLanes="2" speed="13" />
        <!-- Third row -->
            <edge id="2021" from="20" to="21" numLanes="2"
speed="13" />            <edge id="2120" from="21" to="20"
numLanes="2" speed="13" />
            <edge id="2122" from="21" to="22" numLanes="2"
speed="13" />            <edge id="2221" from="22" to="21"
numLanes="2" speed="13" />
    <!-- Verticale edges -->
        <!-- First column -->
            <edge id="0010" from="00" to="10" numLanes="2"
speed="13" />            <edge id="1000" from="10" to="00"
numLanes="2" speed="13" />
            <edge id="1020" from="10" to="20" numLanes="2"
speed="13" />
            <edge id="2010" from="20" to="10" numLanes="2"
speed="13" />
        <!-- Second column -->
            <edge id="0111" from="01" to="11" numLanes="2"
speed="13" />            <edge id="1101" from="11" to="01"
numLanes="2" speed="13" />
```

```
            <edge id="1121" from="11" to="21" numLanes="2"
speed="13" />             <edge id="2111" from="21" to="11"
numLanes="2" speed="13" />
        <!-- Third column -->
            <edge id="0212" from="02" to="12" numLanes="2"
speed="13" />             <edge id="1202" from="12" to="02"
numLanes="2" speed="13" />
            <edge id="1222" from="12" to="22" numLanes="2"
speed="13" />
            <edge id="2212" from="22" to="12" numLanes="2"
speed="13" />
</edges>
```

Appendix B: DTM algorithm code

- dtm/model/dtm-packet.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
#ifndef DTM_PACKET_H
#define DTM_PACKET_H

#include <iostream>
#include "ns3/header.h"
#include "ns3/enum.h"
//#include "ns3/ipv4-address.h"
#include <map>
#include "ns3/nstime.h"

namespace ns3
{
namespace dtm
{

enum MessageType
{
    DTM_HELLO = 1,
DTM_WARNING = 2
};

/**
* \ingroup dtm
* \brief DTM types
*/
class TypeHeader : public Header
{ public:
    /// c-tor
    //TypeHeader ();
    TypeHeader (MessageType t);


    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId();     TypeId
GetInstanceTypeId () const;     uint32_t
GetSerializedSize () const;     void Serialize
(Buffer::Iterator start) const;     uint32_t
Deserialize (Buffer::Iterator start);     void
Print(std::ostream &os) const;
    //\}

    /// Return Type
    MessageType Get () const
    {
        return m_type;
    }
```

```
    /// Check that type is valide
bool IsValid () const
    {
        return m_valid; /// FIXME like in GPSR
    }
    bool operator== (TypeHeader const & o) const; private:
    MessageType m_type;
bool m_valid;
};
std::ostream & operator<< (std::ostream & os, TypeHeader const & h);

/**
* \ingroup dtm
* \brief   Hello Message Format
  \verbatim

  \endverbatim
*/
class HelloHeader : public Header
{ public:
    ///c-tor
  HelloHeader (uint32_t id = 0,
                          uint64_t originPosx = 0,
uint64_t originPosy = 0,
uint64_t speedx = 0,                          uint64_t
speedy = 0);

    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId ();     TypeId
GetInstanceTypeId () const;     uint32_t
GetSerializedSize () const;     void Serialize
(Buffer::Iterator start) const;   uint32_t
Deserialize (Buffer::Iterator start);     void
Print (std::ostream &os) const;
    //\}

    ///\name Fields
//\{
    void SetOriginPosx (uint64_t posx)
    {
        m_originPosx = posx;
    }
    uint64_t GetOriginPosx () const
    {
        return m_originPosx;
    }
    void SetOriginPosy (uint64_t posy)
    {
        m_originPosy = posy;
    }
    uint64_t GetOriginPosy () const
    {
        return m_originPosy;
    }
    void SetId (uint64_t id)
```

```
    {          m_id
= id;      }
    uint64_t GetId () const
    {          return
m_id;      }
    void SetSpeedx (uint64_t speedx)
    {
        m_speedx = speedx;
    }
    uint64_t GetSpeedx () const
    {
        return m_speedx;
    }
    void SetSpeedy (uint64_t speedy)
    {
        m_speedy = speedy;
    }
    uint64_t GetSpeedy () const
    {
        return m_speedy;
    }

    //\}
    bool operator== (HelloHeader const & o) const;
 private:
    uint32_t m_id; // id of source
     uint64_t m_originPosx;
uint64_t m_originPosy;
//uint64_t m_originPosz;
     uint64_t
m_speedx;      uint64_t
m_speedy;
};

/**
* \ingroup dtm
* \brief   Warning Message Format
  \verbatim

  \endverbatim */
class WarningHeader : public Header
{ public:
    /// c-tor
    WarningHeader (uint32_t sourceId = 0,
uint32_t m_prevHopId = 0,
uint32_t m_hopCount = 0,          uint32_t
messageId = 0,
      uint64_t positionx = 0,
uint64_t positiony = 0);

    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId ();      TypeId
GetInstanceTypeId () const;      uint32_t
GetSerializedSize () const;      void Serialize
(Buffer::Iterator start) const;    uint32_t
```

```
Deserialize (Buffer::Iterator start);     void
Print (std::ostream &os) const;
    //\}
    void SetSourceId (uint32_t sourceId)
    {
        m_sourceId =  sourceId;
    }
    uint32_t GetSourceId () const
    {
        return m_sourceId;
    }
    void SetPrevHopId (uint32_t prevHopId)
    {
        m_prevHopId =  prevHopId;
    }
    uint32_t GetPrevHopId () const
    {
        return m_prevHopId;
    }
    void SetHopCount (uint32_t hopCount)
    {
        m_hopCount =  hopCount;
    }
    uint32_t GetHopCount () const
    {
        return m_hopCount;
    }
    void SetMessageId (uint32_t messageId)
    {
        m_messageId = messageId;
    }
    uint32_t GetMessageId () const
    {
        return m_messageId;
    }
    void SetPostionx (uint64_t positionx)
    {
        m_positionx = positionx;
    }
    uint64_t GetPositionx () const
    {
        return m_positionx;
    }
    void SetPositiony (uint64_t positiony)
    {
        m_positiony = positiony;
    }
    uint64_t GetPositiony () const
    {
        return m_positiony;
    }
private:
    uint32_t m_sourceId;
uint32_t m_prevHopId;
uint32_t m_hopCount;     uint32_t
m_messageId;        uint64_t
```

```
m_positionx;     uint64_t
m_positiony;
};

}
}


#endif /* DTMPACKET_H */
```

- dtm/model/dtm-packet.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#include "dtm-packet.h"
#include "ns3/address-utils.h"
#include "ns3/packet.h"
#include "ns3/log.h"

NS_LOG_COMPONENT_DEFINE ("DtmPacket");
 namespace ns3
{ namespace dtm
{

NS_OBJECT_ENSURE_REGISTERED (TypeHeader);

/*TypeHeader::TypeHeader ()
{
    m_valid = true;
}*/

TypeHeader::TypeHeader (MessageType t = DTM_HELLO)
    : m_type (t),
m_valid (true)
{
}

TypeId
TypeHeader::GetTypeId()
{     static TypeId tid = TypeId
("ns3::dtm::TypeHeader")
        .SetParent<Header> ()
        .AddConstructor<TypeHeader> ()
    ;
return tid;
}

TypeId
TypeHeader::GetInstanceTypeId () const
{     return GetTypeId
();
}
uint32_t
TypeHeader::GetSerializedSize () const
{     return
1;
```

```
}
void
TypeHeader::Serialize (Buffer::Iterator start) const
{     start.WriteU8 ((uint8_t)
m_type);
}
uint32_t
TypeHeader::Deserialize (Buffer::Iterator start)
{
    Buffer::Iterator i = start;
uint8_t type = i.ReadU8 ();
m_valid = true;     switch
(type)          {          case
DTM_HELLO:          case
DTM_WARNING:                {
                m_type = (MessageType) type;
break;              }          default:
            m_valid = false;
        }
    uint32_t dist = i.GetDistanceFrom (start);
NS_ASSERT (dist == GetSerializedSize ());     return
dist;
}
void
TypeHeader::Print(std::ostream &os) const
{
    switch (m_type)          {
case DTM_HELLO:             {
os << "HELLO";
break;              }          case
DTM_WARNING:                {
os << "POSITION";
break;              }
default:
            os << "UNKNOWN_TYPE";
        }
}
bool
TypeHeader::operator== (TypeHeader const & o) const
{     return (m_type == o.m_type && m_valid ==
o.m_valid); }  std::ostream &
operator<< (std::ostream & os, TypeHeader const & h){
    h.Print (os);
return os;
}


//-----------------------------------------------------------------------
---------
// HELLO
//-----------------------------------------------------------------------
-------
HelloHeader::HelloHeader (uint32_t id, uint64_t originPosx, uint64_t
originPosy, uint64_t speedx, uint64_t speedy)
  : m_id (id),
```

```
    m_originPosx (originPosx),
m_originPosy (originPosy),
m_speedx (speedx),     m_speedy
(speedy)
{
}

NS_OBJECT_ENSURE_REGISTERED (HelloHeader);

TypeId
HelloHeader::GetTypeId ()
{   static TypeId tid = TypeId
("ns3::dtm::HelloHeader")
    .SetParent<Header> ()
    .AddConstructor<HelloHeader> ()
  ;   return
tid;
}

TypeId
HelloHeader::GetInstanceTypeId () const
{   return GetTypeId
();
}
uint32_t
HelloHeader::GetSerializedSize () const
{   return
36;
}
void
HelloHeader::Serialize (Buffer::Iterator i) const
{
  NS_LOG_DEBUG ("Serialize Id " << m_id
                             << " X " << m_originPosx
                             << " Y " << m_originPosy
                             << " Speed X " << m_speedx
<< " Speed Y " << m_speedy);

  i.WriteHtonU32 (m_id);
  i.WriteHtonU64 (m_originPosx);
  i.WriteHtonU64 (m_originPosy);
  i.WriteHtonU64 (m_speedx);
  i.WriteHtonU64 (m_speedy);
}
uint32_t
HelloHeader::Deserialize (Buffer::Iterator start) {

                  Buffer::Iterator i = start;
                   m_id = i.ReadNtohU32 ();
m_originPosx = i.ReadNtohU64 ();
m_originPosy = i.ReadNtohU64 ();
m_speedx = i.ReadNtohU64 ();
m_speedy = i.ReadNtohU64 ();
```

```
                NS_LOG_DEBUG ("Deserialize Id " << m_id
                                                << " X " << m_originPosx
                                                << " Y " << m_originPosy
                                                << " Speed X " << m_speedx
<< " Speed Y " << m_speedy);                        uint32_t dist =
i.GetDistanceFrom (start);                       NS_ASSERT (dist ==
GetSerializedSize ());                        return dist;
}
void
HelloHeader::Print (std::ostream &os) const
{   os << " Id " <<
m_id
         << " X " << m_originPosx
         << " Y " << m_originPosy
         << " Speed X " << m_speedx
<< " Speed Y " << m_speedy;
}
std::ostream &
operator<< (std::ostream & os, HelloHeader const & h)
{
  h.Print (os);
return os;
}



bool
HelloHeader::operator== (HelloHeader const & o) const
{
  return (m_id == o.m_id &&
                m_originPosx == o.m_originPosx &&
m_originPosy == o.m_originPosy &&
m_speedx == o.m_speedx &&                   m_speedy
== o.m_speedy);
}
//-----------------------------------------------------------------------
---------
// WARNING
//-----------------------------------------------------------------------
---------static TypeId

WarningHeader::WarningHeader (uint32_t sourceId, uint32_t prevHopId,
uint32_t hopCount, uint32_t messageId, uint64_t postionx, uint64_t
positiony)
    : m_sourceId (sourceId),
m_prevHopId (prevHopId),
m_hopCount (hopCount),
m_messageId (messageId),
m_positionx (postionx),
m_positiony (positiony)
{
}

NS_OBJECT_ENSURE_REGISTERED (WarningHeader);

TypeId
```

```
WarningHeader::GetTypeId ()
{   static TypeId tid = TypeId
("ns3::dtm::WarningHeader")
    .SetParent<Header> ()
    .AddConstructor<WarningHeader> ()
  ;   return
tid;
}

TypeId
WarningHeader::GetInstanceTypeId () const
{       return GetTypeId
();
}
uint32_t
WarningHeader::GetSerializedSize () const
{
    return 32;
}
void
WarningHeader::Serialize (Buffer::Iterator start) const {
    NS_LOG_DEBUG ("Serialize Id " << m_sourceId << " MessageId " <<
m_messageId);

  start.WriteHtonU32 (m_sourceId);
  start.WriteHtonU32 (m_prevHopId);
start.WriteHtonU32 (m_hopCount);   start.WriteHtonU32
(m_messageId);   start.WriteHtonU64 (m_positionx);
start.WriteHtonU64 (m_positiony);
}
uint32_t
WarningHeader::Deserialize (Buffer::Iterator start) {
    Buffer::Iterator i = start;
   m_sourceId = i.ReadNtohU32 ();
m_prevHopId = i.ReadNtohU32 ();
m_hopCount = i.ReadNtohU32 ();
m_messageId = i.ReadNtohU32 ();
m_positionx = i.ReadNtohU64 ();
m_positiony = i.ReadNtohU64 ();

  NS_LOG_DEBUG ("Deserialize Id " << m_sourceId << " MessageId " <<
m_messageId);
   uint32_t dist = i.GetDistanceFrom
(start);   NS_ASSERT (dist ==
GetSerializedSize ());   return dist;
}
void
WarningHeader::Print (std::ostream &os) const
{      os << " Id " << m_sourceId << " MessageId " <<
m_messageId; }


}
}
```

- dtm/model/dtm-ptable.h

```
#ifndef DTM_PTABLE_H
#define DTM_PTABLE_H

#include <map>
#include <cassert>
#include <stdint.h>
#include "ns3/ipv4.h"
#include "ns3/timer.h"
#include <sys/types.h>
#include "ns3/node.h"
#include "ns3/node-list.h"
#include "ns3/mobility-model.h"
#include "ns3/vector.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/random-variable-stream.h"
#include <complex>
 namespace ns3
{ namespace dtm
{

/*
* \ingroup dtm
* \brief Position table used by DTM
 */
class PositionTable
{ public:
  /// c-tor
  PositionTable ();

  /**
* \brief Gets the last time the entry was updated
* \param id uint32_t to get time of update from    * \return Time of
  last update to the position
   */
  Time GetEntryUpdateTime (uint32_t id);

  /**
* \brief Adds entry in position table
   */
  void AddEntry (uint32_t id, Vector position, Vector velocity);

  /**
* \brief Deletes entry in position table
   */
  void DeleteEntry (uint32_t id);

  /**
* \brief Gets position from position table
* \param id uint32_t to get position from
* \return Position of that id or NULL if not known
   */
  Vector GetPosition (uint32_t id);

  /**
```

```
* \brief Checks if a node is a neighbour
* \param id uint32_t of the node to check
* \return True if the node is neighbour, false otherwise
   */
  bool isNeighbour (uint32_t id);

  /**
* \brief remove entries with expired lifetime
   */   void
Purge ();

  /**
* \brief clears all entries
   */   void
Clear ();

  /**
* \Get Callback to ProcessTxError
   */
  Callback<void, WifiMacHeader const &> GetTxErrorCallback () const
{     return m_txErrorCallback;
  }
  bool IsInSearch (uint32_t id);

  bool HasPosition (uint32_t id);

  static Vector GetInvalidPosition ()
  {
    return Vector (-1, -1, 0);
  }    double GetThreshold ()
const {     return m_threshold;
  }
  void SetThreshold (double threshold)
  {
    m_threshold = threshold;
  }

  /*Vector GetSpatialDistribution () const {
return m_spatialDist;
  }
  void SetSpatialDistribution (Vector spatialDist)
  {
    m_spatialDist = spatialDist;
  }*/
  void Print (std::ostream & os) const;
 private:
  Time m_entryLifeTime;
  std::map<uint32_t, std::tuple<Vector, Vector, Time>> m_table;
  // TX error callback
  Callback<void, WifiMacHeader const &> m_txErrorCallback;

  double m_threshold;        ///< Distance to mean Threshold
//Vector m_spatialDist;     ///< Spatial Distribution calculated over
neighbor position
```

```
  // Process layer 2 TX error notification
void ProcessTxError (WifiMacHeader const&);


  /// Calculate and update distance Threshold Mc
void CalculateThreshold ();

};  std::ostream & operator<< (std::ostream & os, PositionTable
const & h);

}   // dtm
} // ns3
#endif /* DTM_PTABLE_H */
```

- dtm/model/dtm-ptable.cc

```
#include "dtm-ptable.h"
#include "ns3/simulator.h"
#include "ns3/log.h"
#include <algorithm>
#include <cmath>

NS_LOG_COMPONENT_DEFINE ("DtmTable");
  namespace ns3
{ namespace dtm
{
 /*
  dtm position table
*/

PositionTable::PositionTable ()
{
  m_txErrorCallback = MakeCallback (&PositionTable::ProcessTxError,
this);
  m_entryLifeTime = Seconds (25); //FIXME fix parameter to hello
message timer

  m_threshold = 0;

}

Time
PositionTable::GetEntryUpdateTime (uint32_t id)
{
  if (id == 0)
{
      return Time (Seconds (0));
    }
  std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator i =
m_table.find (id);
  return std::get<2> (i->second);
}

/**
```

```
 * \brief Adds entry in position table and delete earlier entry if
already present
 */
void
PositionTable::AddEntry (uint32_t id, Vector position, Vector
velocity) {
  std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator i =
m_table.find (id);   if (i != m_table.end () || id == (i->first))
    {
      m_table.erase (id);
    }

  m_table.insert (std::make_pair (id, std::make_tuple (position,
velocity, Simulator::Now ())));

  CalculateThreshold ();

}

/**
 * \brief Deletes entry in position table
 */
void
PositionTable::DeleteEntry (uint32_t id)
{
  std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator i =
m_table.find (id);   if (i != m_table.end ())
    {
        m_table.erase (id);
CalculateThreshold ();
    }
}

/**
* \brief Gets position from position table
* \param id uint32_t to get position from
* \return Position of that id or NULL if not known
 */
Vector
PositionTable::GetPosition (uint32_t id)
{

  NodeList::Iterator listEnd = NodeList::End ();
  for (NodeList::Iterator i = NodeList::Begin (); i != listEnd; i++)
    {
      Ptr<Node> node = *i;
if (node->GetId () == id)
        {           return node->GetObject<MobilityModel> ()-
>GetPosition ();
        }
}
  return PositionTable::GetInvalidPosition ();
}

/**
```

```
* \brief Checks if a node is a neighbour
* \param id uint32_t of the node to check
* \return True if the node is neighbour, false otherwise
 */
bool
PositionTable::isNeighbour (uint32_t id)
{   std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator
i = m_table.find (id);
  if (i != m_table.end () || id == (i->first))
    {
      return true;
    }
  return false;
}


/**
 * \brief remove entries with expired lifetime
 */
void
PositionTable::Purge ()
{    if(m_table.empty
())
    {
return;
    }
  std::list<uint32_t> toErase;

  std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator i =
m_table.begin ();
  std::map<uint32_t, std::tuple<Vector, Vector, Time> >::iterator
listEnd = m_table.end ();

  for (; !(i == listEnd); i++)
    {
      if (m_entryLifeTime + GetEntryUpdateTime (i->first) <=
Simulator::Now ())         {           toErase.insert
(toErase.begin (), i->first);

        }
}
  toErase.unique ();
   std::list<uint32_t>::iterator end = toErase.end
();
   for (std::list<uint32_t>::iterator it = toErase.begin (); it !=
end; ++it)
    {

      //m_table.erase (*it);
      PositionTable::DeleteEntry (*it);
    }
}

/**
* \brief clears all entries
```

```
 */
void
PositionTable::Clear ()
{
  m_table.clear ();

  m_threshold = 0;
}

/**
* \ProcessTxError
 */
void
PositionTable::ProcessTxError (WifiMacHeader const & hdr) /// FIXME
complete this function
{
}

/**
* \brief Returns true if is in search for destination
 */
bool
PositionTable::IsInSearch (uint32_t id)
{
  return false;
}
bool
PositionTable::HasPosition (uint32_t id)
{
  return true;
}


/// Calculate and update distance Threshold Mc  void
PositionTable::CalculateThreshold () /// FIXME to complete
{   if (m_table.empty
())
    {
      /// FIXME if needed
    }
  m_threshold = 0.80 - 0.95 * exp(-0.06 * (double) m_table.size ()); }
 /*
void
PositionTable::CalculateSpatialDistribution ()
{
  std::map<uint32_t, std::tuple<Vector,Vector,Time>>::const_iterator i
= m_table.begin ();

  m_spatialDist.x = 0;
m_spatialDist.y = 0;
   if (!(m_table.empty
()))
    {
      double sumx = 0;
double sumy = 0;
```

```
      for (; i != m_table.end (); i++)
        {
          sumx += std::get<0> (i->second).x;
sumy += std::get<0> (i->second).y;
        }        m_spatialDist.x =  ((double) 1/m_table.size
()) * sumx;       m_spatialDist.y =  ((double) 1/m_table.size
()) * sumy;
    }
}
*/
void
PositionTable::Print (std::ostream & os) const
{
  std::map<uint32_t, std::tuple<Vector, Vector, Time>>::const_iterator
i = m_table.begin ();   while (i != m_table.end ())
    {
      os << "\n id : " << i->first;
i++;     }
  os << "\n Threshold : " << m_threshold;
}

std::ostream & operator<< (std::ostream & os, PositionTable const & h)
{
  h.Print(os);
return os;
}

}   // dtm
} // ns3
```

- dtm/model/dtm-wqueue.h

s

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#ifndef DTM_WQUEUE_H
#define DTM_WQUEUE_H

#include <vector>
#include <map>
#include <list>
#include <iostream>

#include "ns3/ipv4-routing-protocol.h"
#include "ns3/simulator.h"
#include "dtm-packet.h" #include
"ns3/enum.h"
#include "ns3/nstime.h"
#include "ns3/vector.h"
  namespace ns3
{ namespace dtm
{

class QueueEntry
{ public:
```

```
    QueueEntry (Vector position = Vector (0.0,0.0,0.0),
        Time backofftime = Seconds (0.0),
Ptr<Packet> packet = Create<Packet> (0),
uint32_t sourceId = 0,          uint32_t
messageId = 0,          uint32_t prevHopId = 0,
uint32_t hopCount = 0,         bool forwarded =
false);

    bool operator== (QueueEntry const & entry) const
    {
        if (m_messageId == entry.GetMessageId ()
            && m_prevHopId == entry.GetPrevHopId ())
            {
                return true;
            }
return false;
    }

    Vector GetPosition () const
    {
        return m_position;
    }
    void SetPosition (Vector position)
    {
        m_position = position;
    }

    Time GetBackOffTime () const
    {
        return m_backOffTime - Simulator::Now ();
    }
    void SetBackOffTime (Time backOffTime)
    {
        m_backOffTime = backOffTime + Simulator::Now ();
    }

    Ptr<Packet> GetPacket () const
    {
        return m_packet;
    }
    void SetPacket (Ptr<Packet> packet)
    {
        m_packet = packet;
    }
    uint32_t GetSourceId () const
    {
        return m_sourceId;
    }
    void SetSourceId (uint32_t sourceId)
    {
        m_sourceId = sourceId;
    }

    uint32_t GetMessageId () const
    {
```

```
        return m_messageId;
    }
    void SetMessageId (uint32_t messageId)
    {
        m_messageId = messageId;
    }
    uint32_t GetPrevHopId () const
    {
        return m_prevHopId;
    }
    void SetPrevHopId (uint32_t prevHopId)
    {
        m_prevHopId = prevHopId;
    }
    uint32_t GetHopCount () const
    {
        return m_hopCount;
    }
    void SetHopCount (uint32_t hopCount)
    {
        m_hopCount = hopCount;
    }
    bool GetForwarded () const
    {
        return m_forwarded;
    }
    void SetForwarded (bool forwarded)
    {
        m_forwarded = forwarded;
    }  private:
Vector m_position;
    Time m_backOffTime;
    Ptr<Packet> m_packet;
uint32_t m_sourceId;
uint32_t m_messageId;
uint32_t m_prevHopId;
uint32_t m_hopCount;     bool
m_forwarded;

};  class
Queue {
public:
    /// Callback <Packet, nodeId>
    Queue ();

    Queue (uint32_t maxLen, Time queueTimeOut)
        : m_maxLen (maxLen),
m_queueTimeOut (queueTimeOut)     {
    }

    /// Add element to Queue
void Add (QueueEntry entry);

    /// Delete messageID elements
void Purge (uint32_t messageId);
```

```
    /// Find if entry already in queue
    bool Find (uint32_t messageId, uint32_t prevId);

    /// Find if messaageId is in queue
bool Exist (uint32_t messageId);

    /// Calculate Spatial Distribution
    Vector CalculateSpatialDist (uint32_t setId);

    Time GetQueueTimeOut () const
    {
        return m_queueTimeOut;
    }
    void SetQueueTimeOut (Time timeOut)
    {
        m_queueTimeOut = timeOut;
    }

    QueueEntry & GetEntry (uint32_t messageId)
    {         return m_queue.find(messageId)-
>second.front ();
    }
    bool IsAlreadyForwarded (uint32_t messageId)
    {
        if (m_queue.find (messageId) != m_queue.end ())
            {                  return m_queue.find
(messageId)->second.front
().GetForwarded ();
}         return
false;
    }
private:
    uint32_t m_maxLen;
Time m_queueTimeOut;

    /// Queue of entry: < message id, < sender id, < sender position,
backofftimer, Packet>>
    std::map<uint32_t, std::list<QueueEntry>> m_queue; };

}
}

#endif
```

- dtm/model/dtm-wqueue.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#include "dtm-wqueue.h"
#include "ns3/log.h"
#include <algorithm>


NS_LOG_COMPONENT_DEFINE ("DtmQueue");
```

```
 namespace ns3
{ namespace dtm
{

/// QueueEntry
QueueEntry::QueueEntry (Vector position, Time backofftime,
        Ptr<Packet> packet,
uint32_t sourceId,          uint32_t
messageId,          uint32_t
prevHopId,          uint32_t
hopCount,         bool forwarded)
:   m_position (position),
m_backOffTime (backofftime),
m_packet (packet),
m_sourceId (sourceId),
m_messageId (messageId),
m_prevHopId (prevHopId),
m_hopCount (hopCount),
m_forwarded (forwarded)
    {
    }


/// Queue
Queue::Queue ()
{
}
void
Queue::Add (QueueEntry entry)
{     m_queue[entry.GetMessageId ()].push_front
(entry); }  void
Queue::Purge (uint32_t messageId)
{     m_queue.erase
(messageId);
}
bool
Queue::Find (uint32_t messageId, uint32_t prevId)
{
    std::map<uint32_t, std::list<QueueEntry>>::iterator i =
m_queue.find (messageId);     if (i != m_queue.end ())
        {
            std::list<QueueEntry>::iterator j = i->second.begin ();
for (; j!=i->second.end (); j++)
                {
                    if (j->GetPrevHopId () == prevId)
                        {
                            return true;
                        }
                }
}
        return false;
}
bool
Queue::Exist (uint32_t messageId)
{
```

```
    if (m_queue.find (messageId) != m_queue.end ())
return true;
     return
false;
}



Vector
Queue::CalculateSpatialDist (uint32_t setId)
{   std::list<QueueEntry> set =
m_queue[setId];

    std::list<QueueEntry>::const_iterator i = set.begin ();

    Vector spatialDist;
   spatialDist.x =
0;   spatialDist.y =
0;

  if (!(set.empty ()))
    {
      double sumx = 0;
double sumy = 0;

      for (; i != set.end (); i++)
        {
          sumx += i->GetPosition ().x;
sumy += i->GetPosition ().y;
        }
      spatialDist.x =  ((double) 1/set.size ()) * sumx;
spatialDist.y =  ((double) 1/set.size ()) * sumy;
}    return spatialDist;
}

}
}
```

Appendix C: KDTM algorithm code

- kdtm/model/kdtm-packet.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */


#ifndef KDTM_PACKET_H
#define KDTM_PACKET_H

#include <iostream>
#include "ns3/header.h"
#include "ns3/enum.h"
//#include "ns3/ipv4-address.h"
#include <map>
#include "ns3/nstime.h"

namespace ns3
{
namespace kdtm
{

enum MessageType
{
    KDTM_HELLO = 1,
KDTM_WARNING = 2
};

/**
* \ingroup kdtm
* \brief kDTM types
*/
class TypeHeader : public Header
{ public:
    /// c-tor
    //TypeHeader ();
    TypeHeader (MessageType t);


    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId();     TypeId
GetInstanceTypeId () const;     uint32_t
GetSerializedSize () const;     void Serialize
(Buffer::Iterator start) const;     uint32_t
Deserialize (Buffer::Iterator start);     void
Print(std::ostream &os) const;
    //\}

    /// Return Type
    MessageType Get () const
    {
        return m_type;
```

```
    }
    /// Check that type is valide
bool IsValid () const
    {
        return m_valid; /// FIXME like in GPSR
    }
    bool operator== (TypeHeader const & o) const; private:
    MessageType m_type;
bool m_valid;
};
std::ostream & operator<< (std::ostream & os, TypeHeader const & h);

/**
* \ingroup kdtm
* \brief   Hello Message Format
  \verbatim

  \endverbatim
*/
class HelloHeader : public Header
{ public:
    ///c-tor
  HelloHeader (uint32_t id = 0,
                          uint64_t originPosx = 0,
uint64_t originPosy = 0,
uint64_t speedx = 0,                              uint64_t
speedy = 0,                             uint64_t
trajectoryBegin = 0,                              uint64_t
beta = 0);

    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId ();     TypeId
GetInstanceTypeId () const;     uint32_t
GetSerializedSize () const;     void Serialize
(Buffer::Iterator start) const;   uint32_t
Deserialize (Buffer::Iterator start);     void
Print (std::ostream &os) const;
    //\}

    ///\name Fields
//\{
    void SetOriginPosx (uint64_t posx)
    {
        m_originPosx = posx;
    }
    uint64_t GetOriginPosx () const
    {
        return m_originPosx;
    }
    void SetOriginPosy (uint64_t posy)
    {
        m_originPosy = posy;
    }
    uint64_t GetOriginPosy () const
```

```
    {         return
m_originPosy;
    }
    void SetId (uint64_t id)
    {         m_id
= id;     }
    uint64_t GetId () const
    {
        return m_id;
    }
    void SetSpeedx (uint64_t speedx)
    {
        m_speedx = speedx;
    }
    uint64_t GetSpeedx () const
    {
        return m_speedx;
    }
    void SetSpeedy (uint64_t speedy)
    {
        m_speedy = speedy;
    }
    uint64_t GetSpeedy () const
    {
        return m_speedy;
    }
    void SetTrajectoryBegin (uint64_t trajectoryBegin)
    {
        m_trajectoryBegin = trajectoryBegin;
    }
    uint64_t GetTrajectoryBegin () const
    {
        return m_trajectoryBegin;
    }
    void SetBeta (uint64_t beta)
    {
        m_beta = beta;
    }
    uint64_t GetBeta () const
    {
        return m_beta;
    }
    //\}
    bool operator== (HelloHeader const & o) const;
 private:      uint32_t m_id; //
id of source
     uint64_t
m_originPosx;     uint64_t
m_originPosy;

    uint64_t m_speedx;
uint64_t m_speedy; uint64_t
m_trajectoryBegin; // time when
its trajectory started
    uint64_t m_beta;  // inverse of average time of trajectory =>
poisson coeff of stability;
```

```
};

/**
* \ingroup kdtm
* \brief   Warning Message Format
  \verbatim

  \endverbatim
*/
class WarningHeader : public Header
{ public:
    /// c-tor
    WarningHeader (uint32_t sourceId = 0,
uint32_t m_prevHopId = 0,
uint32_t m_hopCount = 0,         uint32_t
messageId = 0,       uint64_t positionx =
0,        uint64_t positiony = 0);

    ///\name Header serialization/deserialization
    //\{
    static TypeId GetTypeId ();     TypeId
GetInstanceTypeId () const;     uint32_t
GetSerializedSize () const;     void Serialize
(Buffer::Iterator start) const;   uint32_t
Deserialize (Buffer::Iterator start);     void
Print (std::ostream &os) const;
    //\}
    void SetSourceId (uint32_t sourceId)
    {
        m_sourceId =  sourceId;
    }
    uint32_t GetSourceId () const
    {
        return m_sourceId;
    }
    void SetPrevHopId (uint32_t prevHopId)
    {
        m_prevHopId =  prevHopId;
    }
    uint32_t GetPrevHopId () const
    {
        return m_prevHopId;
    }
    void SetHopCount (uint32_t hopCount)
    {
        m_hopCount =  hopCount;
    }
    uint32_t GetHopCount () const
    {
        return m_hopCount;
    }
    void SetMessageId (uint32_t messageId)
    {
        m_messageId = messageId;
    }
    uint32_t GetMessageId () const
```

```
    {
        return m_messageId;
    }
    void SetPostionx (uint64_t positionx)
    {
        m_positionx = positionx;
    }
    uint64_t GetPositionx () const
    {
        return m_positionx;
    }
    void SetPositiony (uint64_t positiony)
    {
        m_positiony = positiony;
    }
    uint64_t GetPositiony () const
    {
        return m_positiony;
    }
private:
    uint32_t m_sourceId;
uint32_t m_prevHopId;
uint32_t m_hopCount;     uint32_t
m_messageId;        uint64_t
m_positionx;     uint64_t
m_positiony;
};

}
}


#endif /* KDTMPACKET_H */
```

- kdtm/model/kdtm-packet.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
#include "kdtm-packet.h"
#include "ns3/address-utils.h"
#include "ns3/packet.h"
#include "ns3/log.h"

NS_LOG_COMPONENT_DEFINE ("KdtmPacket");
 namespace ns3 {
namespace kdtm {

NS_OBJECT_ENSURE_REGISTERED (TypeHeader);

/*TypeHeader::TypeHeader ()
{
    m_valid = true;
}*/

TypeHeader::TypeHeader (MessageType t = KDTM_HELLO)
```

```
    : m_type (t),
m_valid (true)
{
}

TypeId
TypeHeader::GetTypeId()
{     static TypeId tid = TypeId
("ns3::kdtm::TypeHeader")
        .SetParent<Header> ()
        .AddConstructor<TypeHeader> ()
    ;
return tid;
}

TypeId
TypeHeader::GetInstanceTypeId () const
{     return GetTypeId
();
}
uint32_t
TypeHeader::GetSerializedSize () const
{     return
1;
}
void
TypeHeader::Serialize (Buffer::Iterator start) const
{     start.WriteU8 ((uint8_t)
m_type);
}
uint32_t
TypeHeader::Deserialize (Buffer::Iterator start)
{
    Buffer::Iterator i = start;
uint8_t type = i.ReadU8 ();
m_valid = true;     switch
(type)         {         case
KDTM_HELLO:         case
KDTM_WARNING:              {
                m_type = (MessageType) type;
break;              }         default:
           m_valid = false;
        }
    uint32_t dist = i.GetDistanceFrom (start);
NS_ASSERT (dist == GetSerializedSize ());     return
dist;
}
void
TypeHeader::Print(std::ostream &os) const
{     switch (m_type)
{         case
KDTM_HELLO:
{
                os << "HELLO";
break;              }
case KDTM_WARNING:
```

```
            {
                os << "POSITION";
break;              }
default:
            os << "UNKNOWN_TYPE";
        }
}
bool
TypeHeader::operator== (TypeHeader const & o) const
{     return (m_type == o.m_type && m_valid ==
o.m_valid); }  std::ostream &
operator<< (std::ostream & os, TypeHeader const & h){
    h.Print (os);
return os;
}


//-----------------------------------------------------------------------
---------
// HELLO
//-----------------------------------------------------------------------
---------
HelloHeader::HelloHeader (uint32_t id,
uint64_t originPosx,      uint64_t
originPosy,      uint64_t speedx,
uint64_t speedy,      uint64_t
trajectoryBegin,     uint64_t beta)
: m_id (id),
    m_originPosx (originPosx),
m_originPosy (originPosy),
m_speedx (speedx),     m_speedy
(speedy),
    m_trajectoryBegin (trajectoryBegin),
m_beta (beta)
{
}

NS_OBJECT_ENSURE_REGISTERED (HelloHeader);

TypeId
HelloHeader::GetTypeId ()
{   static TypeId tid = TypeId
("ns3::kdtm::HelloHeader")
    .SetParent<Header> ()
    .AddConstructor<HelloHeader> ()
  ;   return
tid;
}

TypeId
HelloHeader::GetInstanceTypeId () const
{   return GetTypeId
();
}
uint32_t
HelloHeader::GetSerializedSize () const
```

```
{   return
52;
}
void
HelloHeader::Serialize (Buffer::Iterator i) const
{
  NS_LOG_DEBUG ("Serialize Id " << m_id
                            << " X " << m_originPosx
                            << " Y " << m_originPosy
                            << " Speed X " << m_speedx
                            << " Speed Y " << m_speedy
<< " Trajectory Begin Time " << m_trajectoryBegin
                            << " Beta " << m_beta);

  i.WriteHtonU32 (m_id);
  i.WriteHtonU64 (m_originPosx);
  i.WriteHtonU64 (m_originPosy);
  i.WriteHtonU64 (m_speedx);
  i.WriteHtonU64 (m_speedy);
  i.WriteHtonU64 (m_trajectoryBegin);
  i.WriteHtonU64 (m_beta);
}
uint32_t
HelloHeader::Deserialize (Buffer::Iterator start)
{

  Buffer::Iterator i = start;

  m_id = i.ReadNtohU32 ();
m_originPosx = i.ReadNtohU64 ();
m_originPosy = i.ReadNtohU64 ();
m_speedx = i.ReadNtohU64 ();   m_speedy
= i.ReadNtohU64 ();   m_trajectoryBegin
= i.ReadNtohU64 ();   m_beta =
i.ReadNtohU64 ();

  NS_LOG_DEBUG ("Deserialize Id " << m_id
                            << " X " << m_originPosx
                            << " Y " << m_originPosy
                            << " Speed X " << m_speedx
                            <<  "  Speed  Y  "  <<  m_speedy
<< " Trajectory Begin Time " << m_trajectoryBegin
                            << " Beta " << m_beta);
   uint32_t dist = i.GetDistanceFrom
(start);   NS_ASSERT (dist ==
GetSerializedSize ());   return dist;
}
void
HelloHeader::Print (std::ostream &os) const
{   os << " Id " <<
m_id
         << " X " << m_originPosx
         << " Y " << m_originPosy
         << " Speed X " << m_speedx
         << " Speed Y " << m_speedy
         << " Trajectory Begin Time " << m_trajectoryBegin
```

```
        << " Beta " << m_beta;
}
std::ostream &
operator<< (std::ostream & os, HelloHeader const & h)
{
  h.Print (os);
return os;
}


bool
HelloHeader::operator== (HelloHeader const & o) const
{
  return (m_id == o.m_id &&
                m_originPosx == o.m_originPosx &&
m_originPosy == o.m_originPosy &&
m_speedx == o.m_speedx &&                  m_speedy
== o.m_speedy &&
                m_trajectoryBegin == o.m_trajectoryBegin &&
m_beta == o.m_beta);
}
//-----------------------------------------------------------------------
---------
// WARNING
//-----------------------------------------------------------------------
---------static TypeId

WarningHeader::WarningHeader (uint32_t sourceId, uint32_t prevHopId,
uint32_t hopCount, uint32_t messageId, uint64_t postionx, uint64_t
positiony)
    : m_sourceId (sourceId),
m_prevHopId (prevHopId),
m_hopCount (hopCount),
m_messageId (messageId),
m_positionx (postionx),
m_positiony (positiony)
{
}

NS_OBJECT_ENSURE_REGISTERED (WarningHeader);

TypeId
WarningHeader::GetTypeId ()
{   static TypeId tid = TypeId
("ns3::kdtm::WarningHeader")
    .SetParent<Header> ()
    .AddConstructor<WarningHeader> ()
  ;   return
tid;
}

TypeId
WarningHeader::GetInstanceTypeId () const
{       return GetTypeId
();
```

```
}
uint32_t
WarningHeader::GetSerializedSize () const
{
    return 32;
}
void
WarningHeader::Serialize (Buffer::Iterator start) const {
    NS_LOG_DEBUG ("Serialize Id " << m_sourceId << " MessageId " <<
m_messageId);

  start.WriteHtonU32 (m_sourceId);
start.WriteHtonU32 (m_prevHopId);
start.WriteHtonU32 (m_hopCount);   start.WriteHtonU32
(m_messageId);   start.WriteHtonU64 (m_positionx);
start.WriteHtonU64 (m_positiony);
}
uint32_t
WarningHeader::Deserialize (Buffer::Iterator start) {
    Buffer::Iterator i = start;
   m_sourceId = i.ReadNtohU32 ();
m_prevHopId = i.ReadNtohU32 ();
m_hopCount = i.ReadNtohU32 ();
m_messageId = i.ReadNtohU32 ();
m_positionx = i.ReadNtohU64 ();
m_positiony = i.ReadNtohU64 ();

  NS_LOG_DEBUG ("Deserialize Id " << m_sourceId << " MessageId " <<
m_messageId);
   uint32_t dist = i.GetDistanceFrom
(start);   NS_ASSERT (dist ==
GetSerializedSize ());   return dist;
}
void
WarningHeader::Print (std::ostream &os) const
{      os << " Id " << m_sourceId << " MessageId " <<
m_messageId; }


}
}
```

- kdtm/model/kdtm-ptable.h

```
#ifndef KDTM_PTABLE_H
#define KDTM_PTABLE_H

#include <map>
#include <cassert>
#include <stdint.h>
#include "ns3/ipv4.h"
#include "ns3/timer.h"
#include <sys/types.h>
#include "ns3/node.h"
```

```
#include "ns3/node-list.h"
#include "ns3/mobility-model.h"
#include "ns3/vector.h"
#include "ns3/wifi-mac-header.h"
#include "ns3/random-variable-stream.h"
#include <complex>
 namespace ns3 {
namespace kdtm {

/*
* \ingroup kdtm
* \brief Position table used by kDTM
 */
class PositionTable
{ public:
  /// c-tor
  PositionTable ();
  PositionTable (double maxRange, Vector position, Vector velocity);
/**
* \brief Gets the last time the entry was updated
* \param id uint32_t to get time of update from    * \return Time of
  last update to the position
   */
  Time GetEntryUpdateTime (uint32_t id);

  /**
* \brief Adds entry in position table
   */
  void AddEntry (uint32_t id, Vector position, Vector velocity, Time
time, double Betaj, Time tj);

  /**
* \brief Deletes entry in position table
   */
  void DeleteEntry (uint32_t id);

  /**
* \brief Gets position from position table
* \param id uint32_t to get position from
* \return Position of that id or NULL if not known
   */
  Vector GetPosition (uint32_t id);

  /**
* \brief Checks if a node is a neighbour
* \param id uint32_t of the node to check
* \return True if the node is neighbour, false otherwise
   */
  bool isNeighbour (uint32_t id);

  /**
* \brief remove entries with expired lifetime
   */   void
Purge ();
```

```
  /**
* \brief clears all entries
   */   void
Clear ();

  /**
* \Get Callback to ProcessTxError
   */
  Callback<void, WifiMacHeader const &> GetTxErrorCallback () const
  {
    return m_txErrorCallback;
  }

  /**
* \brief Calculate distance Threshold Mc based on Position predicaition
  algorithm
   */
  double CalculateThreshold (Time time);
bool IsInSearch (uint32_t id);

  bool HasPosition (uint32_t id);

  static Vector GetInvalidPosition ()
  {
    return Vector (-1, -1, 0);
  }    double GetMaxRange ()
const {     return m_maxRange;
  }
  void SetMaxRange (double maxRange)
  {
    m_maxRange = maxRange;
  }

  Vector GetMyPosition () const {
return m_myPosition;
  }
  void SetMyPosition (Vector position)
  {
    m_myPosition = position;
  }

  Vector GetMyVelocity () const {
return m_myVelocity;
  }
  void SetMyVelocity (Vector velocity)
  {
    m_myVelocity = velocity;
  }    double GetPoissonCoeff ()
const {     return
m_poissonCoeff.second;
  }
  void SetPoissonCoeff (double time)
  {
```

```
    m_poissonCoeff.second = (m_poissonCoeff.first *
m_poissonCoeff.second + time) / (m_poissonCoeff.first + 1);
m_poissonCoeff.first++;
  }

  Time GetTrajectoryBegin () const {
return m_trajectoryBegin;
  }
  void SetTrajectoryBegin (Time time)
  {
    m_trajectoryBegin = time;
  }
  double GetAlpha () const {
    return m_alpha;
  }
  void SetAlpha (double alpha)
  {
    m_alpha = alpha;
  }
  void Print (std::ostream & os);

  double CalculateDegree (Time time);
 private:
  Time m_entryLifeTime;
  //  map: node Id  <Position, velocity, time_from, Time_to, Betaj, tj>
  std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time>> m_table;
  // TX error callback
  Callback<void, WifiMacHeader const &> m_txErrorCallback;

  Vector m_myPosition;
  Vector m_myVelocity;

  double m_maxRange;

  double m_alpha;

  std::pair<uint32_t, double> m_poissonCoeff;

  Time m_trajectoryBegin;

  // Process layer 2 TX error notification
void ProcessTxError (WifiMacHeader const&);

  /// Calucale link power equation Pij(t) = Aij*t^2 + Bij*t + Cij
double CalculateAij (Vector velocity);
  double CalculateBij (Vector position, Vector velocity);
double CalculateCij (Vector position);

  /// Calculate time tij(to) and tij(from)
  std::pair<Time, Time> CalculateTimeFromTo (Time time, Vector
position, Vector velocity);

  /// Calculate stability of liaison ij: pij(t)
  double CalculateStability (double time, double tj, double Betaj);
```

```
  /// Calculate degree of liason ij: Degij(t)
  double CalculateDoubleSigmoid (double t_from, double t_to, double
t);
};
std::ostream & operator<< (std::ostream & os, PositionTable & h);
} // kdtm
} // ns3
#endif /* kDTM_PTABLE_H */
```

• kdtm/model/kdtm-ptable.cc

```
#include "kdtm-ptable.h"
#include "ns3/simulator.h"
#include "ns3/log.h"
#include <algorithm>
#include <cmath>

NS_LOG_COMPONENT_DEFINE ("KdtmTable");
  namespace ns3
{ namespace kdtm
{
 /*
  kdtm position table
*/
PositionTable::PositionTable ()
{
}

PositionTable::PositionTable (double maxRange, Vector position, Vector
velocity)
{
  NS_LOG_INFO (" Kdtm table constructor ");

  m_txErrorCallback = MakeCallback (&PositionTable::ProcessTxError,
this);
  m_entryLifeTime = Seconds (25); //FIXME fix parameter to hello
message timer

  m_maxRange = maxRange;
   m_myPosition =
position;   m_myVelocity =
velocity;

  m_poissonCoeff = std::make_pair(1,300.0);
   m_alpha =
10.0;
}

/**
 * \brief Adds entry in position table and delete earlier entry if
already present
 */
void
```

```
PositionTable::AddEntry (uint32_t id, Vector position, Vector
velocity, Time time, double Betaj, Time tj)
{   std::map<uint32_t, std::tuple<Vector, Vector, Time, Time,
double, Time> >::iterator i = m_table.find (id);   if (i !=
m_table.end () || id == (i->first))
    {
      m_table.erase (id);
    }
  std::pair<Time, Time> times_from_to = CalculateTimeFromTo (time,
position, velocity);

  m_table.insert (std::make_pair (id,
std::make_tuple (position,
velocity,
times_from_to.first,
times_from_to.second,
Betaj,                          tj)));

}

/**
 * \brief Deletes entry in position table
 */
void
PositionTable::DeleteEntry (uint32_t id)
{   std::map<uint32_t, std::tuple<Vector, Vector, Time, Time,
double,
Time> >::iterator i = m_table.find (id);
if (i != m_table.end ())
    {
        m_table.erase (id);
    }
}

/**
* \brief Gets position from position table
* \param id uint32_t to get position from
* \return Position of that id or NULL if not known
 */
Vector
PositionTable::GetPosition (uint32_t id)
{

  NodeList::Iterator listEnd = NodeList::End ();
  for (NodeList::Iterator i = NodeList::Begin (); i != listEnd; i++)
{
      Ptr<Node> node = *i;
if (node->GetId () == id)
        {                 return  node->GetObject<MobilityModel>  ()-
>GetPosition ();          }     }
  return PositionTable::GetInvalidPosition ();

}

/**
```

```
* \brief Checks if a node is a neighbour
* \param id uint32_t of the node to check
* \return True if the node is neighbour, false otherwise
 */ bool
Position
Table::i
sNeighbo
ur
(uint32_
t id)
{
 std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time> >::iterator i = m_table.find (id);   if (i != m_table.end ()
|| id == (i->first))
    {
      return true;
    }
  return false;
}

Time
PositionTable::GetEntryUpdateTime (uint32_t id)
{   std::map<uint32_t, std::tuple<Vector, Vector, Time, Time,
double,
Time> >::iterator i = m_table.find (id);
return std::get<3> (i->second);
}

/**
 * \brief remove entries with expired lifetime
 */
void
PositionTable::Purge ()
{    if(m_table.empty
())
    {
return;
    }
  std::list<uint32_t> toErase;

  std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time> >::iterator i = m_table.begin ();
  std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time> >::iterator listEnd = m_table.end ();

  for (; !(i == listEnd); i++)
    {
      if (GetEntryUpdateTime (i->first) <= Simulator::Now ())
        {           toErase.insert (toErase.begin (),
i->first);
        }
}
  toErase.unique ();

  std::list<uint32_t>::iterator end = toErase.end ();
```

```
   for (std::list<uint32_t>::iterator it = toErase.begin (); it !=
end; ++it)
    {

      //m_table.erase (*it);
      PositionTable::DeleteEntry (*it);
    }
}

/**
* \brief clears all entries
 */
void
PositionTable::Clear ()
{   m_table.clear
();
}

/**
* \ProcessTxError
 */
void
PositionTable::ProcessTxError (WifiMacHeader const & hdr) /// FIXME
complete this function
{
}

/**
* \brief Returns true if is in search for destination
 */
bool
PositionTable::IsInSearch (uint32_t id)
{
  return false;
}
bool
PositionTable::HasPosition (uint32_t id)
{
  return true;
}

/**
 * Calculate Threshold based on Distance To Mean method and Kinetic
graph method  */ double
PositionTable::CalculateThreshold (Time time)
{   double kinetic_degree = CalculateDegree
(time);
   return 0.80 - 0.95 * exp(-0.06 *
kinetic_degree);
  //return kinetic_degree;

}
double
PositionTable::CalculateDegree (Time time)
```

```
{   Purge ();   if
(m_table.empty ())
{       return 0;     }
double stability;
double degree;

  double kinetic_degree = 0;

  std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time>>::const_iterator i = m_table.begin ();
for (; i != m_table.end (); i++)
    {
      stability = CalculateStability (time.GetSeconds (),
std::get<4> (i->second),
std::get<5> (i-
>second).GetSeconds ());

      NS_LOG_INFO (" Time: " << time
        << " Beta i: " << 1/m_poissonCoeff.second
        << " Beta j: " << std::get<4> (i->second)
        << " ti: " << m_trajectoryBegin
        << " tj: " << std::get<5> (i->second)
        << " Stability: " << stability);

      degree = CalculateDoubleSigmoid (std::get<2> (i-
>second).GetSeconds (),
std::get<3> (i-
>second).GetSeconds (),
time.GetSeconds ());

      NS_LOG_INFO (" Degree: " << degree);

      kinetic_degree += stability * degree;
    }

  NS_LOG_INFO (" Kinetic Degree: " << kinetic_degree);

  return kinetic_degree;
}

void
PositionTable::Print (std::ostream & os)
{   Purge
();
  std::map<uint32_t, std::tuple<Vector, Vector, Time, Time, double,
Time>>::const_iterator i = m_table.begin ();
while (i != m_table.end ())
    {
      os << "\n id : " << i->first
      << " time arrived " << std::get<2> (i->second).GetSeconds ()
      << " time before leave " << std::get<3> (i->second).GetSeconds
();
i++;
    }
}
```

```
// Private functions
//{


/// Calculate element Aij of equation Pij(t) = Aij*t^2 + Bij*t + Cij
double
PositionTable::CalculateAij (Vector velocity)
{
  return pow (m_myVelocity.x - velocity.x, 2) + pow (m_myVelocity.y -
velocity.y, 2);
}

/// Calculate element Bij of equation Pij(t) = Aij*t^2 + Bij*t + Cij
double
PositionTable::CalculateBij (Vector position, Vector velocity)
{
  return 2 * (m_myPosition.x - position.x) * (m_myVelocity.x -
velocity.x)
       + 2 * (m_myPosition.y - position.y) * (m_myVelocity.y -
velocity.y);
}

/// Calculate element Cij of equation Pij(t) = Aij*t^2 + Bij*t + Cij
double
PositionTable::CalculateCij (Vector position)
{
  return pow (m_myPosition.x - position.x, 2) + pow (m_myPosition.y -
position.y, 2);
}

/// Calculate neighbors time in and out. Solve the equation Pij(t) = 0
std::pair<Time, Time>
PositionTable::CalculateTimeFromTo (Time time, Vector position, Vector
velocity) {
  double Aij = CalculateAij (velocity);   double
Bij = CalculateBij (position, velocity);   double
Cij = CalculateCij (position);
   double
from;   double
to;

  // Time infinity must be > to simulation time
double infinity =  500.0;

  if (Aij == 0)
    {
      if (Bij == 0)
        {
          //NS_LOG_INFO (" Aij: " << Aij << " Bij: " << Bij << " Cij:
" << Cij);
          //NS_LOG_INFO (" Time from: " << Seconds (0.0) << " Time to:
" << Seconds (500));
           return std::make_pair (time, Seconds
(infinity));         }
```

```
      from = - (Cij - pow (m_maxRange, 2)) / Bij;
to = - (Cij - pow (m_maxRange, 2)) / Bij;

      //NS_LOG_INFO (" Aij: " << Aij << " Bij: " << Bij << " Cij: " <<
Cij);
      //NS_LOG_INFO (" Time from: " << from << " Time to: " << to);
      NS_LOG_INFO ("aij = bij = 0, t_from " <<  (time.GetSeconds () +
from)
        << " t_to " <<  (time.GetSeconds () + to));

      return std::make_pair (time + Seconds (from), time + Seconds
(to));
    }
  double delta = pow (Bij, 2) - 4 * Aij * (Cij - pow(m_maxRange, 2));

  if (delta > 0)
    {
      from = (- Bij - sqrt (delta)) / (2 * Aij);
if (from > 0)          {           to = from;
          from = (- Bij + sqrt (delta)) / (2 * Aij);
        }
else
{
          to = (- Bij + sqrt (delta)) / (2 * Aij);
        }
}
  if (delta == 0)
    {
      from = - Bij / (2 * Aij);
to = - Bij / (2 * Aij);
     }   if
(delta < 0)
    {       from =
0;       to =
infinity;
    }    if (time.GetSeconds () +
from < 0)
    {       from = -
(time.GetSeconds ());
    }

    NS_LOG_INFO (" t_from " << (time.GetSeconds () + from)
      << " t_to " << (time.GetSeconds () + to));

  return std::make_pair (Seconds (time.GetSeconds () + from),
                         Seconds (time.GetSeconds () + to)); }


/// Calculate double Sigmoid of liason ij with time (t_to and t_from)
double
PositionTable::CalculateDoubleSigmoid (double t_from, double t_to,
double t) {   return (1.0 / (1.0 + exp (- m_alpha * (t - t_from)))) *
(1.0 / (1.0
+ exp (m_alpha * (t - t_to))));
```

```
}

/** Calculate stability coefficient of liason ij, pij(t):
 *  pij(t)= exp (-(Betai + Betaj)*(t - (ti*Betai + tj*Betaj)/(Betai +
Betaj)))
*/ double
PositionTable::CalculateStability (double time, double tj, double
Betaj) {
  double Betai = 1.0 / m_poissonCoeff.second;
double ti = m_trajectoryBegin.GetSeconds ();

  if (Betaj == 0.0 && Betai == 0.0)
    {
return 1;
    }

  NS_LOG_INFO (" Betai + Betaj: " << Betai);
   return exp (-(Betai + Betaj)*(time - ((ti*Betai + tj*Betaj)/(Betai
+
Betaj))));
}

//}

/// Opreators
//{
std::ostream & operator<< (std::ostream & os, PositionTable & h)
{
  h.Print(os);
return os;
}

//}

}   // kdtm
} // ns3
```

• kdtm/model/kdtm-wqueue.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#ifndef KDTM_WQUEUE_H
#define KDTM_WQUEUE_H

#include <vector>
#include <map>
#include <list>
#include <iostream>

#include "ns3/ipv4-routing-protocol.h"
#include "ns3/simulator.h"
#include "kdtm-packet.h"
#include "ns3/enum.h"
#include "ns3/nstime.h"
```

```
#include "ns3/vector.h"
  namespace ns3
{ namespace kdtm
{

class QueueEntry
{ public:
    QueueEntry (Vector position = Vector (0.0,0.0,0.0),
        Time backofftime = Seconds (0.0),
Ptr<Packet> packet = Create<Packet> (0),
uint32_t sourceId = 0,          uint32_t
messageId = 0,          uint32_t prevHopId = 0,
uint32_t hopCount = 0,         bool forwarded =
false);

    bool operator== (QueueEntry const & entry) const
    {
        if (m_messageId == entry.GetMessageId ()
&& m_prevHopId == entry.GetPrevHopId ())
            {
                return true;
            }
return false;
    }

    Vector GetPosition () const
    {
        return m_position;
    }
    void SetPosition (Vector position)
    {
        m_position = position;
    }

    Time GetBackOffTime () const
    {
        return m_backOffTime - Simulator::Now ();
    }
    void SetBackOffTime (Time backOffTime)
    {
        m_backOffTime = backOffTime + Simulator::Now ();
    }

    Ptr<Packet> GetPacket () const
    {
        return m_packet;
    }
    void SetPacket (Ptr<Packet> packet)
    {
        m_packet = packet;
    }
    uint32_t GetSourceId () const
    {
        return m_sourceId;
    }
```

```
    void SetSourceId (uint32_t sourceId)
    {
        m_sourceId = sourceId;
    }
    uint32_t GetMessageId () const
    {
        return m_messageId;
    }
    void SetMessageId (uint32_t messageId)
    {
        m_messageId = messageId;
    }
    uint32_t GetPrevHopId () const
    {
        return m_prevHopId;
    }
    void SetPrevHopId (uint32_t prevHopId)
    {
        m_prevHopId = prevHopId;
    }
    uint32_t GetHopCount () const
    {
        return m_hopCount;
    }
    void SetHopCount (uint32_t hopCount)
    {
        m_hopCount = hopCount;
    }      bool GetForwarded
() const
    {
        return m_forwarded;
    }
    void SetForwarded (bool forwarded)
    {
        m_forwarded = forwarded;
    }  private:
Vector m_position;
    Time m_backOffTime;
    Ptr<Packet> m_packet;
uint32_t m_sourceId;     uint32_t
m_messageId;     uint32_t
m_prevHopId;     uint32_t
m_hopCount;     bool m_forwarded;

};  class
Queue {
public:
    /// Callback <Packet, nodeId>
    Queue ();

    Queue(uint32_t maxLen, Time queueTimeOut)
        : m_maxLen (maxLen),
            m_queueTimeOut (queueTimeOut)
    {
    }
```

```
    /// Add element to Queue
void Add (QueueEntry entry);

    /// Delete messageID elements
void Purge (uint32_t messageId);

    /// Find if entry already in queue
    bool Find (uint32_t messageId, uint32_t prevId);

    /// Find if entry exist     bool
Exist (uint32_t messageId);

    /// Calculate Spatial Distribution
    Vector CalculateSpatialDist (uint32_t setId);

    Time GetQueueTimeOut () const
    {
        return m_queueTimeOut;
    }
    void SetQueueTimeOut (Time timeOut)
    {
        m_queueTimeOut = timeOut;
    }

    QueueEntry & GetEntry (uint32_t messageId)
    {         return m_queue.find(messageId)-
>second.front ();
    }      bool IsAlreadyForwarded (uint32_t
messageId)
    {
        if (m_queue.find (messageId) != m_queue.end ())
            {                  return m_queue.find
(messageId)->second.front
().GetForwarded ();
}         return
false;
    }

private:
    uint32_t m_maxLen;
Time m_queueTimeOut;

    /// Queue of entry: < message id, < sender id, < sender position,
backofftimer, Packet>>
    std::map<uint32_t, std::list<QueueEntry>> m_queue; };

}
}

#endif
```

- kdtm/model/kdtm-wqueue.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

#include "kdtm-wqueue.h"
#include "ns3/log.h"
#include <algorithm>


NS_LOG_COMPONENT_DEFINE ("KdtmQueue");
 namespace ns3 {
namespace kdtm {

/// QueueEntry
QueueEntry::QueueEntry (Vector position, Time backofftime,
        Ptr<Packet> packet,
uint32_t sourceId,          uint32_t
messageId,          uint32_t
prevHopId,          uint32_t
hopCount,         bool forwarded)
:   m_position (position),
m_backOffTime (backofftime),
m_packet (packet),
m_sourceId (sourceId),
m_messageId (messageId),
m_prevHopId (prevHopId),
m_hopCount (hopCount),
m_forwarded (forwarded)
    {
    }


/// Queue
Queue::Queue ()
{
}
void
Queue::Add (QueueEntry entry)
{     m_queue[entry.GetMessageId ()].push_front
(entry); }  void
Queue::Purge (uint32_t messageId)
{     m_queue.erase
(messageId);
}
bool
Queue::Find (uint32_t messageId, uint32_t prevId)
{
    std::map<uint32_t, std::list<QueueEntry>>::iterator i =
m_queue.find (messageId);     if (i != m_queue.end ())
        {
            std::list<QueueEntry>::iterator j = i->second.begin ();
for (; j!=i->second.end (); j++)
                {
                    if (j->GetPrevHopId () == prevId)
                        {
                            return true;
                        }
```

```
                }
}
        return false;
}
bool
Queue::Exist (uint32_t messageId)
{
    if (m_queue.find (messageId) != m_queue.end ())
return true;
     return
false;
}


Vector
Queue::CalculateSpatialDist (uint32_t setId)
{   std::list<QueueEntry> set =
m_queue[setId];

    std::list<QueueEntry>::const_iterator i = set.begin ();
    Vector spatialDist;
   spatialDist.x =
0;   spatialDist.y =
0;

  if (!(set.empty ()))
    {
      double sumx = 0;
double sumy = 0;

      for (; i != set.end (); i++)
        {
          sumx += i->GetPosition ().x;
sumy += i->GetPosition ().y;
        }        spatialDist.x =  ((double) 1/set.size
()) * sumx;       spatialDist.y =  ((double) 1/set.size
()) * sumy;
    }
  return spatialDist;
}

}
}
```

Appendix D: DTM ns3 main for urban and highway scenario

```
// Basic C++ library
#include <iostream>
#include <fstream>
#include <sstream>

#include "ns3/vector.h"
#include "ns3/string.h"
#include "ns3/socket.h"
#include "ns3/double.h"
#include "ns3/config.h"
#include "ns3/log.h"
#include "ns3/command-line.h"
#include "ns3/position-allocator.h"


// Mobility Helper modules
#include "ns3/core-module.h"
#include "ns3/mobility-module.h"

// needed to work with SUMO config files
#include "ns3/ns2-mobility-helper.h"

// for fix mobility
#include "ns3/mobility-model.h"
#include "ns3/mobility-helper.h"

// Wave handler inclusion
#include "ns3/wave-net-device.h"
#include "ns3/wave-mac-helper.h"

// Wifi 802_11p handler inclusion
#include "ns3/ocb-wifi-mac.h"
#include "ns3/wifi-80211p-helper.h"
#include "ns3/wave-helper.h"

// Ipv4 and internet stack helper includes
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/ipv4-interface-container.h"

// Wifi simulator model include #include
"ns3/yans-wifi-helper.h"

// needed for animation
#include "ns3/netanim-module.h"

#include "ns3/dtm-packet.h"
#include "ns3/dtm-wqueue.h"
#include "ns3/dtm-ptable.h"
 using namespace
ns3;
```

```
// debug log definebbb
NS_LOG_COMPONENT_DEFINE ("UrbanModelScript");

// define global position table and message queue for nodes
dtm::PositionTable * m_neighbors; dtm::Queue * m_queue;

dtm::PositionTable *
InitializeNeighbors (uint32_t size)
{    return new
dtm::PositionTable[size];
}
dtm::Queue *
InitializeQueue (uint32_t size)
{    return new
dtm::Queue[size];
}

uint32_t m_messageId = 0;

// Set up the global broadcast address
uint16_t beacon_port = 80; ///
Differents times
const Time m_beacon_interval = Seconds (15);
const Time m_purge_interval = Seconds (0.50);
const Time m_tmax = Seconds (0.05); const
double m_maxRange = 300.0;

InetSocketAddress broadcastAddr = InetSocketAddress (Ipv4Address
("255.255.255.255"), beacon_port);

NodeContainer nodes;
NetDeviceContainer devices;

// Measures variables
std::vector<uint32_t> m_receivers;
std::vector<uint32_t> m_rebroadcast;
uint32_t m_warningBytes = 0; uint32_t
m_helloBytes = 0;


// Prints actual position and velocity when a course change event
occurs static void
CourseChange (std::ostream *os, std::string foo, Ptr<const
MobilityModel> mobility)
{
   //NS_LOG_INFO ("CourseChange");

   Vector pos = mobility->GetPosition (); // Get position
   Vector vel = mobility->GetVelocity (); // Get velocity

   Ptr<const Object> object = mobility;
   Ptr<Node> node = object->GetObject<Node> ();

   // Prints position and velocities
```

```
   *os << Simulator::Now () << " POS: x=" << pos.x << ", y=" << pos.y
<< ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y
   << ", z=" << vel.z << std::endl;
}


/// Calculate the Distance to mean for the node and its neighbors double
CalculateDTM (Ptr<Node> node, Vector spatialDist, double maxRangeArea)
{
   //NS_LOG_LOGIC (" Distance To Mean function: ");

   //NS_LOG_INFO (" spatialDist.X: " << spatialDist.x << "
spatialDist.Y: " << spatialDist.y);

   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();

   Vector position = mobility->GetPosition ();
   //NS_LOG_INFO (" position.X: " << position.x << " position.Y: " <<
position.y);
     double M = (1/maxRangeArea) * sqrt (pow (position.x
- spatialDist.x, 2) + pow (position.y - spatialDist.y,
2));

   //NS_LOG_INFO (" Distance To Mean : " << M);

   return M;
}
void PurgeQueue (uint32_t nodeId, uint32_t messageId)
{    m_queue[nodeId].Purge
(messageId);
}
void Forward (Ptr<Node> recvNode, dtm::QueueEntry& entry)
{    if (!(entry == m_queue[recvNode->GetId
()].GetEntry
(entry.GetMessageId ())))
return;

   Vector spatialDist = m_queue[recvNode->GetId
()].CalculateSpatialDist (entry.GetMessageId ());
   double dtm = CalculateDTM (recvNode, spatialDist, m_maxRange);
double threshold = m_neighbors[recvNode->GetId ()].GetThreshold ();

   if (dtm <= threshold) {
      Simulator::Schedule (m_queue[recvNode->GetId ()].GetQueueTimeOut
(), &PurgeQueue, recvNode->GetId (), entry.GetMessageId ());
return;
   }

   // Update nb of rebroadcast
   m_rebroadcast.at (entry.GetMessageId ()) += 1;

   //NS_LOG_INFO( " Id: " << recvNode->GetId () << " Forward packet: "
<< entry.GetMessageId ());
   TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
```

```
   Ptr<Socket> forwardSocket = Socket::CreateSocket (recvNode, tid);
forwardSocket->Connect (broadcastAddr);    forwardSocket-
>SetAllowBroadcast (true);

   Ptr<Packet> packet = entry.GetPacket ();

   // Get mobility of receiver node
   Ptr<Object> object = recvNode;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();


   // Create and Add Warning Header
   dtm::WarningHeader wHeader = dtm::WarningHeader (
entry.GetSourceId (),       recvNode->GetId (),
entry.GetHopCount ()+1,       entry.GetMessageId (),
      (uint64_t)mobility->GetPosition ().x,       (uint64_t)mobility-
>GetPosition ().y
      );
   packet->AddHeader (wHeader);

   // Create and Add Type Header
   dtm::TypeHeader tHeader = dtm::TypeHeader (dtm::DTM_WARNING);
packet->AddHeader (tHeader);

   // indicate entry as forwarded
   m_queue[recvNode->GetId ()].GetEntry (entry.GetMessageId
()).SetForwarded (true);
   //NS_LOG_INFO (" FORWARDED MODIFIED: " << m_queue[recvNode->GetId
()].IsAlreadyForwarded (entry.GetMessageId ()));

   // update bytes send
   m_warningBytes += packet->GetSize ();

   // Rebroadcast Packet    forwardSocket-
>Send (packet);

   // Close Socket    forwardSocket-
>Close ();

   // Declare entry as forwarded
   Simulator::Schedule (m_queue[recvNode->GetId ()].GetQueueTimeOut
(), &PurgeQueue, recvNode->GetId (), entry.GetMessageId ()); }

Time CalculateBackOffTime (Vector recvPosition, Vector senderPosition,
double maxRangeArea, Time tmax)
{
   //NS_LOG_LOGIC(" CalculateBackOffTime: ");
    double distance = sqrt ( pow (recvPosition.x -
senderPosition.x,2)
+ pow (recvPosition.y - senderPosition.y,2));
double coeff = (1.0 -(double)
distance/maxRangeArea);

   return Seconds (coeff*tmax.GetSeconds ());
}
```

```
void GenerateHelloMessage (Ptr<Node> node) {
   //NS_LOG_INFO ("GenerateHelloMessage");

   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();
if (mobility == 0)
      {
         NS_FATAL_ERROR ("The requested mobility model is not a
mobility model");       }    else
      {
         Ptr<Packet> packet = Create<Packet> ();

         uint32_t id = node->GetId ();
         Vector position = mobility->GetPosition ();
         Vector speed = mobility->GetVelocity ();

         /*
         NS_LOG_INFO (" Id " << id
                        << " X " << position.x
                        << " Y " << position.y
                        << " Speed X " << speed.x
<< " Speed Y " << speed.y);
         */
         dtm::HelloHeader hHeader =  dtm::HelloHeader (id,
(uint64_t) position.x,
                                        (uint64_t) position.y,
                                        (uint64_t) speed.x,
                                        (uint64_t) speed.y);

         dtm::TypeHeader tHeader = dtm::TypeHeader (dtm::DTM_HELLO);

         packet->AddHeader (hHeader);          packet-
>AddHeader (tHeader);
          m_neighbors[id].Purge
();

          // Beacon source Socket
         TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
         Ptr<Socket> socket = Socket::CreateSocket (node, tid);

         socket->Connect (broadcastAddr);          socket-
>SetAllowBroadcast (true);

         // update bytes send
         m_helloBytes += packet->GetSize ();
          socket->Send
(packet);

         socket->Close ();
         Simulator::Schedule (m_beacon_interval,
&GenerateHelloMessage, node);
      }
}
void GenerateWarningMessage (Ptr<Node> node)
{
```

```
   //NS_LOG_INFO (" GenerateWarningMessage ");

   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();
if (mobility == 0)
      {
         NS_FATAL_ERROR ("The requested mobility model is not a
mobility model");       }    else
      {
         TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");

         Ptr<Packet> packet = Create<Packet> (200);

         Vector position = mobility->GetPosition ();
         //NS_LOG_INFO (" MessageId: " << m_messageId);
         dtm::WarningHeader wHeader = dtm::WarningHeader(node->GetId
(),                                              node->GetId
(),
                                                 0,
m_messageId++,
position.x,
position.y);
         dtm::TypeHeader tHeader = dtm::TypeHeader (dtm::DTM_WARNING);

         packet->AddHeader (wHeader);          packet-
>AddHeader (tHeader);

         Ptr<Socket> socket = Socket::CreateSocket (node, tid);
socket->Connect (broadcastAddr);          socket->SetAllowBroadcast
(true);

         // update bytes send
         m_warningBytes += packet->GetSize ();
          socket-
>Send(packet);
          socket->Close
();

         //Create reachability variable
m_receivers.insert (m_receivers.end (), 0);
m_rebroadcast.insert (m_rebroadcast.end (), 0);       }
}
void ReceivePacket (Ptr<Socket> socket) {
   //NS_LOG_INFO ("ReceivePacket");

   Ptr<Node> recvNode = socket->GetNode ();
   Ptr<Packet> packet = socket->Recv();

   dtm::TypeHeader tHeader (dtm::DTM_HELLO);
packet->RemoveHeader (tHeader);    if
(!tHeader.IsValid ())
      {
      //NS_LOG_DEBUG ("dtm message " << packet->GetUid () << " with
unknown type received: " << tHeader.Get () << ". Ignored");
return;       }
```

```
   switch (tHeader.Get ())
      {          case
(dtm::DTM_HELLO):
            {
               dtm::HelloHeader hHeader;
packet->RemoveHeader (hHeader);

               uint32_t id;
Vector position;
Vector speed;

               id = hHeader.GetId ();
               position.x = hHeader.GetOriginPosx ();
position.y = hHeader.GetOriginPosy ();
speed.x = hHeader.GetSpeedx ();                speed.y
= hHeader.GetSpeedy ();

               /*
               NS_LOG_INFO("Receive Hello:");
               NS_LOG_INFO (" Id " << id
                        << " X " << position.x
                        << " Y " << position.y
                        << " Speed X " << speed.x
<< " Speed Y " << speed.y);

               NS_LOG_INFO(" \n Neighbors of :" << recvNode->GetId ()
                     << " :: " << m_neighbors[recvNode->GetId ()]);
               */
               m_neighbors[recvNode->GetId ()].AddEntry (id,
position,                                                    speed);
                break;             }
case (dtm::DTM_WARNING):
            {// To Code
               dtm::WarningHeader wHeader = dtm::WarningHeader ();
packet->RemoveHeader (wHeader);

               //NS_LOG_INFO("Receive Warning: " <<
wHeader.GetMessageId () << " from: " << wHeader.GetPrevHopId ());
               // if receiver node == source node
               if (recvNode->GetId () == wHeader.GetSourceId ())
{
                     //NS_LOG_INFO ("packet ignored: I am source
node");                     return;
                  }

               // if received message already received
if (m_queue[recvNode->GetId ()].Find
(wHeader.GetMessageId (), wHeader.GetPrevHopId ()))
                  {
                     //NS_LOG_INFO ("packet ignored: already
received");                        return;
                  }

               // if packet already forwarded by this node
if (m_queue[recvNode->GetId ()].IsAlreadyForwarded
```

```
(wHeader.GetMessageId ()))
                  {
                     //NS_LOG_INFO ("packet ignored: already
forwarded");
                     return;
                  }
                if (! (m_queue[recvNode->GetId
()].Exist
(wHeader.GetMessageId ())))
                  {
                     // Update nb of bytes received and receivers
m_receivers.at (wHeader.GetMessageId ()) += 1;
                  }

               Ptr<Object> object = recvNode;
               Ptr<MobilityModel> mobility = object-
>GetObject<MobilityModel> ();

               // Calculate backoff time
               Time backoffTime = CalculateBackOffTime (
mobility->GetPosition (),
                  Vector((double)wHeader.GetPositionx (),
(double)wHeader.GetPositiony (), 0),
m_maxRange,                    m_tmax
                  );
                dtm::QueueEntry en = dtm::QueueEntry (
Vector((double)wHeader.GetPositionx (),
(double)wHeader.GetPositiony (), 0),
backoffTime + Simulator::Now (),
packet,                    wHeader.GetSourceId (),
wHeader.GetMessageId (),
wHeader.GetPrevHopId (),
wHeader.GetHopCount (),                    false);

               m_queue[recvNode->GetId ()].Add (en);
Simulator::Schedule (en.GetBackOffTime (), &Forward, recvNode, en);
break;             }          default:
         NS_LOG_DEBUG (" dtm message " << packet->GetUid () << " with
unknown type received: " << tHeader.Get () << ". Ignored");       }
}

NodeContainer CreateNodes (std::string traceFile, uint32_t nodeNum) {

   // Create all nodes.
   //NS_LOG_INFO ("Creating Topology"); <= for Urban scenario

   NodeContainer nodes;
nodes.Create (nodeNum);

   // Create Ns2MobilityHelper with the specified trace log file as
parameter
   //Ns2MobilityHelper ns2 = Ns2MobilityHelper (traceFile);
//ns2.Install (); // configure movements for each node, while reading
trace file   <= for urban scenario
```

```
   MobilityHelper mobility;   <= for highway scenario

   mobility.SetPositionAllocator
("ns3::RandomRectanglePositionAllocator",
      "X", StringValue
("ns3::UniformRandomVariable[Min=0.0|Max=1000.0]"),
      "Y", StringValue
("ns3::UniformRandomVariable[Min=0.0|Max=10.0]"));

   mobility.SetMobilityModel ("ns3::ConstantVelocityMobilityModel");
    mobility.Install
(nodes);
        return
nodes;
}

NetDeviceContainer CreateDevice (NodeContainer nodes) {
   NetDeviceContainer devices;

   // Create Wave handler and network model
   // Create Wifi network model physical layer
   //YansWavePhyHelper wavePhy =  YansWavePhyHelper::Default ();
YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
wifiPhy.Set ("TxPowerStart", DoubleValue (25));  // 250-300 meter
transmission range
   wifiPhy.Set ("TxPowerEnd", DoubleValue (25));  // 250-300 meter
transmission range
   wifiPhy.Set ("TxPowerLevels",UintegerValue (1));
wifiPhy.Set ("TxGain",DoubleValue (0));    wifiPhy.Set
("RxGain",DoubleValue (0));
   wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (-101.0));

   // Create Wifi network model
   YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default
();

   // Create wave default wifichannel and apply it to network physical
layer
   wifiPhy.SetChannel (wifiChannel.Create ());

   // Set wifi using ieee wifi802.11p Helper
   wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11);

   NqosWaveMacHelper wifi80211pMac = NqosWaveMacHelper::Default ();
   Wifi80211pHelper wifi80211p = Wifi80211pHelper::Default ();

   // Define Wifi Remote Station Manager
std::string phyMode ("OfdmRate6MbpsBW10MHz");
   wifi80211p.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                       "DataMode",StringValue
(phyMode),
                                       "ControlMode",StringValue
(phyMode));
```

```
   // Create set of nodes with the right network characteristics
//devices = waveHelper.Install (wavePhy, waveMac, nodes);    devices
= wifi80211p.Install (wifiPhy, wifi80211pMac, nodes);


   // Enable physical level tracing
   wifiPhy.EnablePcap ("wave-simple-80211p", devices);

   return devices;
}
int main (int argc, char *argv[]) {

   //LogComponentEnable ("Ns2MobilityHelper", LOG_ALL);
    std::string
traceFile;    std::string
logFile;
    int nodeNum;
double duration;

   // Setting attributes
   //traceFile = "scratch/urban-model/ns2-
mobilitymodel/urban.mobility.tcl";    //nodeNum =
20;    duration = 300.0;
   logFile  =  "scratch/urban-model/logs/ns2-mobility-trace.txt";
// Enable logging from the ns2 helper
   LogComponentEnable ("Ns2MobilityHelper",LOG_LEVEL_DEBUG);

   // Parse command line attribute
CommandLine cmd;
   cmd.AddValue ("n", "Number of nodes", nodeNum);
cmd.Parse (argc,argv);

   nodes = CreateNodes (traceFile, nodeNum);

   // Create net device.
   devices = CreateDevice (nodes);

   // Set internet stack helper (set ipv4)
InternetStackHelper internet;    internet.Install
(nodes);

   Ipv4AddressHelper ipv4;
   NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.0.0", "255.255.0.0");
   Ipv4InterfaceContainer ipv4Container = ipv4.Assign (devices);
   TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");

   // Create tab of Position Table
   m_neighbors = InitializeNeighbors (devices.GetN ());
   // Create tab of queue
   m_queue = InitializeQueue (devices.GetN ());

   // Register callback function when receiving
for (uint32_t i=0; i != devices.GetN (); ++i)
```

```
   {
      // Get node position
      Ptr<Object> object = nodes.Get (i);
      Ptr<ConstantVelocityMobilityModel> mobility = object-
>GetObject<ConstantVelocityMobilityModel> ();

      mobility->SetVelocity (Vector (20.0+ (i%10), 0.0, 0.0));  <= for
highway scenario only

      // Initialize table for each node
m_neighbors[i] = dtm::PositionTable ();

      // Initialize queue for each node
m_queue[i] = dtm::Queue (50, Seconds(10));
      // Beacon sink on every nodes
      Ptr<Socket> recv = Socket::CreateSocket (nodes.Get (i), tid);
recv->Bind (InetSocketAddress (nodes.Get (i)
                    ->GetObject<ns3::Ipv4> ()
                    ->GetAddress (1,0)
                     .GetLocal (), beacon_port));

      NS_LOG_INFO ("Addresse local: " << nodes.Get
(i)>GetObject<ns3::Ipv4> ()->GetAddress (1,0).GetLocal ());
      //device.allocate
      //device->SetReceiveCallback( MakeCallback( &WaveNetDevice))
recv->SetRecvCallback (MakeCallback (&ReceivePacket));

      Simulator::ScheduleWithContext (i, Seconds (1.0 + i*0.1),
&GenerateHelloMessage, nodes.Get (i));
   }

   uint32_t randN;
   for (uint32_t mes = 50; mes < 250; mes+=5)
      {
         randN = rand () % nodes.GetN ();
         Simulator::ScheduleWithContext(randN, Seconds(mes),
&GenerateWarningMessage, nodes.Get (randN));
      }

   // open log file for output
std::ofstream os;    os.open
(logFile.c_str ());

   // Configure callback for logging
   Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
               MakeBoundCallback (&CourseChange, &os));

   // Create netanim animation file
   AnimationInterface anim ("animation.xml");

   Simulator::Stop (Seconds (duration));
   Simulator::Run ();
   Simulator::Destroy ();
```

```
   // Calculate res
double reach_mean = 0.0;
   std::vector<uint32_t>::iterator recv = m_receivers.begin ();
for (; recv != m_receivers.end (); recv++)
      {
         reach_mean += *recv;
      }
   reach_mean = reach_mean/((double) m_receivers.size ()*nodeNum);

   double rebroad_mean = 0.0;

   for (uint32_t rebr = 0; rebr < m_rebroadcast.size (); rebr++)
      {
         if (m_receivers.at(rebr) != 0) {
            NS_LOG_INFO (" rebr: " << rebr
               << " m_rebr " << m_rebroadcast.at (rebr)
<< " m_recv " << m_receivers.at (rebr));
rebroad_mean += m_rebroadcast.at (rebr) / (double)
m_receivers.at (rebr);
         }
}
   rebroad_mean = rebroad_mean/(double) m_rebroadcast.size ();
   NS_LOG_INFO ("\n nb of nodes " << nodeNum
      << "\n Reachability " << reach_mean
      << "\n rebroadcast per covered nodes " << rebroad_mean
      << "\n nb of warning bytes transmitted " << m_warningBytes
      << "\n nb of hello bytes transmitted " << m_helloBytes);

   // close log file
os.close();
    return
0;
}
```

Appendix E: KDTM ns3 main for for urban and highway scenario

```
// Basic C++ library
#include <iostream>
#include <fstream>
#include <sstream>

#include "ns3/vector.h"
#include "ns3/string.h"
#include "ns3/socket.h"
#include "ns3/double.h"
#include "ns3/config.h"
#include "ns3/log.h"
#include "ns3/command-line.h"
#include "ns3/position-allocator.h"


// Mobility Helper modules
#include "ns3/core-module.h"
#include "ns3/mobility-module.h"

// needed to work with SUMO config files
#include "ns3/ns2-mobility-helper.h"

// for fix mobility
#include "ns3/mobility-model.h"
#include "ns3/mobility-helper.h"

// Wave handler inclusion
#include "ns3/wave-net-device.h"
#include "ns3/wave-mac-helper.h"

// Wifi 802_11p handler inclusion
#include "ns3/ocb-wifi-mac.h"
#include "ns3/wifi-80211p-helper.h"
#include "ns3/wave-helper.h"

// Ipv4 and internet stack helper includes
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/ipv4-interface-container.h"

// Wifi simulator model include #include
"ns3/yans-wifi-helper.h"

// needed for animation
#include "ns3/netanim-module.h"

#include "ns3/kdtm-packet.h"
#include "ns3/kdtm-wqueue.h"
#include "ns3/kdtm-ptable.h"
```

```
#include "ns3/system-mutex.h"
 using namespace
ns3; // debug log
definebbb
NS_LOG_COMPONENT_DEFINE ("UrbanModelScript");

// define global position table and message queue for nodes
kdtm::PositionTable * m_neighbors; kdtm::Queue * m_queue;

kdtm::PositionTable *
InitializeNeighbors (uint32_t size)
{    return new
kdtm::PositionTable[size];
}
kdtm::Queue *
InitializeQueue (uint32_t size)
{    return new
kdtm::Queue[size];
}
uint32_t m_messageId = 0;

// Set up the global broadcast address
uint16_t beacon_port = 80; ///
Differents times
const Time m_beacon_interval = Seconds (15);
const Time m_purge_interval = Seconds (0.50);
const Time m_tmax = Seconds (0.05); const
double m_maxRange = 300.0;

InetSocketAddress broadcastAddr = InetSocketAddress (Ipv4Address
("255.255.255.255"), beacon_port);

InetSocketAddress
InitializeBroadcastAddresse ()
{
   return InetSocketAddress (Ipv4Address ("255.255.255.255"),
beacon_port);
}



NodeContainer nodes;
NetDeviceContainer devices;


// Measures variables
std::vector<uint32_t> m_receivers;
std::vector<uint32_t> m_rebroadcast;
uint32_t m_warningBytes = 0; uint32_t
m_helloBytes = 0;

 //{ static
void
CourseChange (std::ostream *os, std::string foo, Ptr<const
```

```
MobilityModel> mobility);

double
CalculateDTM (Ptr<Node> node, Vector spatialDist, double
maxRangeArea);

void
PurgeQueue (uint32_t nodeId, uint32_t messageId);

void
Forward (Ptr<Node> recvNode, kdtm::QueueEntry& entry);

Time
CalculateBackOffTime (Vector recvPosition, Vector senderPosition,
double maxRangeArea, Time tmax);

void
GenerateHelloMessage (Ptr<Node> node);

void
GenerateWarningMessage (Ptr<Node> node);

void
ReceivePacket (Ptr<Socket> socket);

NodeContainer
CreateNodes (std::string traceFile, uint32_t nodeNum);

NetDeviceContainer
CreateDevice (NodeContainer nodes);

void
AdvertisePosition (Vector old_position, Vector old_velocity, Ptr<Node>
node);
//}
void
AdvertisePosition (Vector old_position, Vector old_velocity, Ptr<Node>
node)
{
   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();
   Vector new_position = mobility->GetPosition ();
   Vector new_velocity = mobility->GetVelocity ();

   double d = sqrt (pow(old_position.x - new_position.x,2) + pow
(old_position.y - new_position.y,2));
   //double n = 2.0/3.0;
double n = 2.0;
     if (d >
n*m_maxRange)
      {
         GenerateHelloMessage (node);
         Simulator::Schedule (Seconds(10), &AdvertisePosition,
new_position, new_velocity, node);
      }
else       {
```

```
         Simulator::Schedule (Seconds(10), &AdvertisePosition,
old_position, old_velocity, node);
      }
}
void
AdvertisePositionDegree (Ptr<Node> node, double old_deg)
{
   double new_deg = m_neighbors[node->GetId ()].CalculateDegree
(Simulator::Now ());

   if (old_deg > new_deg)
      {
         GenerateHelloMessage (node);
      }
   Simulator::Schedule (Seconds(30), &AdvertisePositionDegree, node,
new_deg);

}

// Prints actual position and velocity when a course change event
occurs static void
CourseChange (std::ostream *os, std::string foo, Ptr<const
MobilityModel> mobility)
{
   //NS_LOG_INFO ("CourseChange");

   Vector pos = mobility->GetPosition (); // Get position
   Vector vel = mobility->GetVelocity (); // Get velocity

   Ptr<const Object> object = mobility;
   Ptr<Node> node = object->GetObject<Node> ();

   Time tbegin = m_neighbors[node->GetId ()].GetTrajectoryBegin (); //
Get beginning of previous trajectory change

   // Update fields in each node
   m_neighbors[node->GetId ()].SetPoissonCoeff (Simulator::Now
().GetSeconds () - tbegin.GetSeconds ());
   m_neighbors[node->GetId ()].SetTrajectoryBegin (Simulator::Now ());

   // Prints position and velocities
   *os << Simulator::Now ().GetSeconds () << " node: " << node->GetId
() << " POS: x=" << pos.x << ", y=" << pos.y
   << ", z=" << pos.z << "; VEL:" << vel.x << ", y=" << vel.y
   << ", z=" << vel.z << std::endl;
}

/// Calculate the Distance to mean for the node and its neighbors double
CalculateDTM (Ptr<Node> node, Vector spatialDist, double maxRangeArea)
{
   //NS_LOG_LOGIC (" Distance To Mean function: ");

   //NS_LOG_INFO (" spatialDist.X: " << spatialDist.x << "
spatialDist.Y: " << spatialDist.y);
```

```
   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();

   Vector position = mobility->GetPosition ();
   //NS_LOG_INFO (" position.X: " << position.x << " position.Y: " <<
position.y);
     double M = (1/maxRangeArea) * sqrt (pow (position.x
- spatialDist.x, 2) + pow (position.y - spatialDist.y,
2));

   //NS_LOG_INFO (" Distance To Mean : " << M);

   return M;
}
void
PurgeQueue (uint32_t nodeId, uint32_t messageId)
{    m_queue[nodeId].Purge
(messageId);
}
void
Forward (Ptr<Node> recvNode, kdtm::QueueEntry& entry)
{    if (!(entry == m_queue[recvNode->GetId
()].GetEntry
(entry.GetMessageId ())))
return;

   Vector spatialDist = m_queue[recvNode->GetId
()].CalculateSpatialDist (entry.GetMessageId ());
   double kDTM = CalculateDTM (recvNode, spatialDist, m_maxRange);
double threshold = m_neighbors[recvNode->GetId
()].CalculateThreshold (Simulator::Now ());

   //NS_LOG_INFO (" DTM: " << kDTM
    //  << " threshold: " << threshold);

   if (kDTM <= threshold) {
      Simulator::Schedule (m_queue[recvNode->GetId ()].GetQueueTimeOut
(), &PurgeQueue, recvNode->GetId (), entry.GetMessageId ());
return;
   }

   // Update nb of rebroadcast
   m_rebroadcast.at (entry.GetMessageId ()) += 1;

   /*
   NS_LOG_INFO( " Id: " << recvNode->GetId ()
      << " Forward packet: " << entry.GetMessageId ()
      << " nb of hop " << m_rebroadcast.at (entry.GetMessageId ()));
*/

   TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
   Ptr<Socket> forwardSocket = Socket::CreateSocket (recvNode, tid);
forwardSocket->Connect (broadcastAddr);    forwardSocket-
>SetAllowBroadcast (true);
```

```
   Ptr<Packet> packet = entry.GetPacket ();

   // Get mobility of receiver node
   Ptr<Object> object = recvNode;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();


   // Create and Add Warning Header
   kdtm::WarningHeader wHeader = kdtm::WarningHeader (
entry.GetSourceId (),       recvNode->GetId (),
entry.GetHopCount ()+1,       entry.GetMessageId (),
      (uint64_t)mobility->GetPosition ().x,       (uint64_t)mobility-
>GetPosition ().y
      );
   packet->AddHeader (wHeader);

   // Create and Add Type Header
   kdtm::TypeHeader tHeader = kdtm::TypeHeader (kdtm::KDTM_WARNING);
packet->AddHeader (tHeader);

   // indicate entry as forwarded
   m_queue[recvNode->GetId ()].GetEntry (entry.GetMessageId
()).SetForwarded (true);
   //NS_LOG_INFO (" FORWARDED MODIFIED: " << m_queue[recvNode->GetId
()].IsAlreadyForwarded (entry.GetMessageId ()));

   // update bytes send
   m_warningBytes += packet->GetSize ();

   // Rebroadcast Packet    forwardSocket-
>Send (packet);

   // Close Socket    forwardSocket-
>Close ();

   // Declare entry as forwarded
   Simulator::Schedule (m_queue[recvNode->GetId ()].GetQueueTimeOut
(), &PurgeQueue, recvNode->GetId (), entry.GetMessageId ()); }

Time
CalculateBackOffTime (Vector recvPosition, Vector senderPosition,
double maxRangeArea, Time tmax)
{
   //NS_LOG_LOGIC(" CalculateBackOffTime: ");
    double distance = sqrt ( pow (recvPosition.x -
senderPosition.x,2)
+ pow (recvPosition.y - senderPosition.y,2));
double coeff = (1.0 -(double)
distance/maxRangeArea);

   return Seconds (coeff*tmax.GetSeconds ());
}
void
GenerateHelloMessage (Ptr<Node> node)
{
```

```
   //NS_LOG_INFO ("GenerateHelloMessage");

   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();
if (mobility == 0)
      {
         //NS_FATAL_ERROR ("The requested mobility model is not a
mobility model");       }    else
      {
         Ptr<Packet> packet = Create<Packet> (200);

         uint32_t id = node->GetId ();
         Vector position = mobility->GetPosition ();
         Vector speed = mobility->GetVelocity ();

         Time ti = m_neighbors[node->GetId ()].GetTrajectoryBegin ();
double inv_beta = m_neighbors[node->GetId ()].GetPoissonCoeff
();          double beta = (1.0 / inv_beta) *
10000.0;

         /*NS_LOG_INFO (" Id " << id
                        << " X " << position.x
                        << " Y " << position.y
                        << " Speed X " << speed.x
                        << " Speed Y " << speed.y
                        << " Trajectory Begin Time " << ti
                        << " Trajectory Average Time " << inv_beta);
         */
         kdtm::HelloHeader hHeader =  kdtm::HelloHeader (id,
(uint64_t) position.x,
                                      (uint64_t) position.y,
                                      (uint64_t) speed.x,
                                      (uint64_t) speed.y,
                                      (uint64_t) ti.GetInteger (),
                                      (uint64_t) beta);
          kdtm::TypeHeader tHeader =
kdtm::TypeHeader
(kdtm::KDTM_HELLO);

         packet->AddHeader (hHeader);          packet-
>AddHeader (tHeader);

         m_neighbors[id].Purge ();

          // Beacon source Socket
         TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
         Ptr<Socket> socket = Socket::CreateSocket (node, tid);
socket->Connect (broadcastAddr);          socket->SetAllowBroadcast
(true);

         /*NS_LOG_INFO (" hello to send " << sizeof (* packet)
            << " socket " << socket->GetAllowBroadcast ());
         */

         // update bytes send
```

```
         m_helloBytes += packet->GetSize ();
          socket->Send
(packet);
          socket->Close
();
      }
}
void
GenerateWarningMessage (Ptr<Node> node)
{
   //NS_LOG_INFO (" GenerateWarningMessage ");

   Ptr<Object> object = node;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();
if (mobility == 0)
      {
         NS_FATAL_ERROR ("The requested mobility model is not a
mobility model");       }    else
      {
         TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");

         Ptr<Packet> packet = Create<Packet> ();

         Vector position = mobility->GetPosition ();
         //NS_LOG_INFO (" MessageId: " << m_messageId);
         kdtm::WarningHeader wHeader = kdtm::WarningHeader(node->GetId
(),                                              node->GetId
(),
                                                 0,
m_messageId++,
position.x,
position.y);          kdtm::TypeHeader tHeader =
kdtm::TypeHeader
(kdtm::KDTM_WARNING);

         packet->AddHeader (wHeader);          packet-
>AddHeader (tHeader);

         Ptr<Socket> socket = Socket::CreateSocket (node, tid);
socket->Connect (broadcastAddr);          socket->SetAllowBroadcast
(true);

         // update bytes send
         m_warningBytes += packet->GetSize ();
          socket-
>Send(packet);
          socket->Close
();

         //Create reachability variable
m_receivers.insert (m_receivers.end (), 0);
m_rebroadcast.insert (m_rebroadcast.end (), 0);       }
}
void
ReceivePacket (Ptr<Socket> socket)
```

```
{
  // NS_LOG_INFO ("ReceivePacket");

   Ptr<Node> recvNode = socket->GetNode ();
   Ptr<Packet> packet = socket->Recv();

   Ptr<Object> object = recvNode;
   Ptr<MobilityModel> mobility = object->GetObject<MobilityModel> ();

   kdtm::TypeHeader tHeader (kdtm::KDTM_HELLO);
packet->RemoveHeader (tHeader);    if
(!tHeader.IsValid ())
      {
      //NS_LOG_DEBUG ("kDTM message " << packet->GetUid () << " with
unknown type received: " << tHeader.Get () << ". Ignored");
return;       }
   switch (tHeader.Get ())
      {          case
(kdtm::KDTM_HELLO):
            {
               kdtm::HelloHeader hHeader;
packet->RemoveHeader (hHeader);

               uint32_t id;
Vector position;
               Vector speed;
Time ti;                double
beta;

               id = hHeader.GetId ();
               position.x = hHeader.GetOriginPosx ();
position.y = hHeader.GetOriginPosy ();
speed.x = hHeader.GetSpeedx ();                speed.y =
hHeader.GetSpeedy ();               ti = Seconds
(hHeader.GetTrajectoryBegin ());               beta =
hHeader.GetBeta () / 10000.0;

               m_neighbors[recvNode->GetId ()].SetMyPosition
(mobility->GetPosition ());
m_neighbors[recvNode->GetId ()].SetMyVelocity
(mobility->GetVelocity ());

               /*
               NS_LOG_INFO("Receive Hello:");
               NS_LOG_INFO (" Id " << id
                        << " X " << position.x
                        << " Y " << position.y
                        << " Speed X " << speed.x
                        << " Speed Y " << speed.y
                        << " Trajectory begin time " << ti
                        << " Beta " << beta);
               NS_LOG_INFO (" my Position X " << m_neighbors[recvNode-
>GetId ()].GetMyPosition ().x
                  << " my Position Y " << m_neighbors[recvNode->GetId
()].GetMyPosition ().y
```

```
                  << " my Speed x " << m_neighbors[recvNode->GetId
()].GetMyVelocity ().x);
*/
               m_neighbors[recvNode->GetId ()].AddEntry (id,
position,                                                  speed,
                                                Simulator::Now (),
beta,                                                   ti);

               /*
               NS_LOG_INFO(" \n Neighbors of :" << recvNode->GetId ()
                     << " :: " << m_neighbors[recvNode->GetId ()]
                     << "\n Threshold: " << m_neighbors[recvNode-
>GetId ()].CalculateThreshold (Simulator::Now ()));
               */

               //Simulator::Schedule (m_neighbors[recvNode->GetId
()].GetEntryUpdateTime (id) - Simulator::Now () -Seconds (1.0),
//            &GenerateHelloMessage, recvNode);
                 break;             }
case (kdtm::KDTM_WARNING):
            {// To Code
               kdtm::WarningHeader wHeader = kdtm::WarningHeader ();
packet->RemoveHeader (wHeader);

               //NS_LOG_INFO("Receive Warning: " <<
wHeader.GetMessageId () << " from: " << wHeader.GetPrevHopId ());

               // if receiver node == source node
               if (recvNode->GetId () == wHeader.GetSourceId ())
                  {
                     //NS_LOG_INFO ("packet ignored: I am source
node");                       return;
                  }

               // if received message already received
if (m_queue[recvNode->GetId ()].Find
(wHeader.GetMessageId (), wHeader.GetPrevHopId ()))
                  {
                     //NS_LOG_INFO ("packet ignored: already
received");                       return;
                  }

               // if packet already forwarded by this node
if (m_queue[recvNode->GetId ()].IsAlreadyForwarded
(wHeader.GetMessageId ()))
                  {
                     //NS_LOG_INFO ("packet ignored: already
forwarded");                       return;
                  }
                if (! (m_queue[recvNode->GetId
()].Exist
(wHeader.GetMessageId ())))
                  {
                     // Update nb of bytes received and receivers
m_receivers.at (wHeader.GetMessageId ()) += 1;
```

```
                }

              // Calculate backoff time
              Time backoffTime = CalculateBackOffTime (
mobility->GetPosition (),
                Vector((double)wHeader.GetPositionx (),
(double)wHeader.GetPositiony (), 0),
m_maxRange,                    m_tmax
                );
               kdtm::QueueEntry en = kdtm::QueueEntry (
Vector((double)wHeader.GetPositionx (),
(double)wHeader.GetPositiony (), 0),
backoffTime + Simulator::Now (),
packet,
                wHeader.GetSourceId (),
wHeader.GetMessageId (),
wHeader.GetPrevHopId (),
wHeader.GetHopCount (),                    false);
              m_queue[recvNode->GetId ()].Add
(en);

              Simulator::Schedule (en.GetBackOffTime (), &Forward,
recvNode, en);                                  break;             }
default:
         NS_LOG_DEBUG (" kDTM message " << packet->GetUid () << " with
unknown type received: " << tHeader.Get () << ". Ignored");       }
}

NodeContainer
CreateNodes (std::string traceFile, uint32_t nodeNum)
{

   // Create all nodes.
   //NS_LOG_INFO ("Creating Topology"); <= for urban scenario

   NodeContainer nodes;
nodes.Create (nodeNum);

   // Create Ns2MobilityHelper with the specified trace log file as
parameter
   //Ns2MobilityHelper ns2 = Ns2MobilityHelper (traceFile);
//ns2.Install (); // configure movements for each node, while reading
trace file   <= for urban scenario


   MobilityHelper mobility;      <= for highway scenario

   mobility.SetPositionAllocator
("ns3::RandomRectanglePositionAllocator",
      "X", StringValue
("ns3::UniformRandomVariable[Min=0.0|Max=1000.0]"),
      "Y", StringValue
("ns3::UniformRandomVariable[Min=0.0|Max=10.0]"));

   mobility.SetMobilityModel ("ns3::ConstantVelocityMobilityModel");
```

```
   mobility.Install (nodes);

   return nodes;
}

NetDeviceContainer
CreateDevice (NodeContainer nodes)
{
   NetDeviceContainer devices;

   // Create Wave handler and network model
   // Create Wifi network model physical layer
   //YansWavePhyHelper wavePhy =  YansWavePhyHelper::Default ();
YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
wifiPhy.Set ("TxPowerStart", DoubleValue (25));  // 250-300 meter
transmission range
   wifiPhy.Set ("TxPowerEnd", DoubleValue (25));  // 250-300 meter
transmission range
   wifiPhy.Set ("TxPowerLevels",UintegerValue (1));
wifiPhy.Set ("TxGain",DoubleValue (0));    wifiPhy.Set
("RxGain",DoubleValue (0));
   wifiPhy.Set ("EnergyDetectionThreshold", DoubleValue (-101.0));
   // Create Wifi network model
   YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default
();

   // Create wave default wifichannel and apply it to network physical
layer
   wifiPhy.SetChannel (wifiChannel.Create ());
// Set wifi using ieee wifi802.11p Helper
   wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11);

   NqosWaveMacHelper wifi80211pMac = NqosWaveMacHelper::Default ();
   Wifi80211pHelper wifi80211p = Wifi80211pHelper::Default ();

   // Define Wifi Remote Station Manager
std::string phyMode ("OfdmRate6MbpsBW10MHz");
   wifi80211p.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                       "DataMode",StringValue
(phyMode),
                                       "ControlMode",StringValue
(phyMode));


   // Create set of nodes with the right network characteristics
//devices = waveHelper.Install (wavePhy, waveMac, nodes);    devices
= wifi80211p.Install (wifiPhy, wifi80211pMac, nodes);


   // Enable physical level tracing
   wifiPhy.EnablePcap ("wave-simple-80211p", devices);

   return devices;
```

```
}
int
main (int argc, char *argv[])
{

   //LogComponentEnable ("Ns2MobilityHelper", LOG_ALL);
    std::string
traceFile;    std::string
logFile;
    int nodeNum;
double duration;

   // Setting attributes
   //traceFile = "scratch/urban-model/ns2-
mobilitymodel/urban.mobility.tcl";    //nodeNum =
20;    duration = 300.0;
   logFile = "scratch/urban-model/logs/ns2-mobility-trace.txt";
   // Enable logging from the ns2 helper
   LogComponentEnable ("Ns2MobilityHelper",LOG_LEVEL_DEBUG);
   // Parse command line attribute
CommandLine cmd;
   cmd.AddValue ("n", "Number of nodes", nodeNum);
cmd.Parse (argc,argv);


   // initialize broadcast addresse
broadcastAddr = InitializeBroadcastAddresse ();
// Enable logging from the ns2 helper
   //LogComponentEnable ("Ns2MobilityHelper",LOG_LEVEL_DEBUG);

   nodes = CreateNodes (traceFile, nodeNum);

   // Create net device.
   devices = CreateDevice (nodes);

   // Set internet stack helper (set ipv4)
InternetStackHelper internet;    internet.Install
(nodes);

   Ipv4AddressHelper ipv4;
   //NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.0.0", "255.255.0.0");
   Ipv4InterfaceContainer ipv4Container = ipv4.Assign (devices);
   TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");

   // Create tab of Position Table
   m_neighbors = InitializeNeighbors (devices.GetN ());
   // Create tab of queue
   m_queue = InitializeQueue (devices.GetN ());

   // Register callback function when receiving
for (uint32_t i=0; i != devices.GetN (); ++i)
   {
      // Get node position
      Ptr<Object> object = nodes.Get (i);
```

```
      Ptr<ConstantVelocityMobilityModel> mobility = object-
>GetObject<ConstantVelocityMobilityModel> ();

      mobility->SetVelocity (Vector (20.0+ (i % 10), 0.0, 0.0));    <= for
highway scenario


      // Initialize table for each node
      m_neighbors[i] = kdtm::PositionTable (m_maxRange, mobility-
>GetPosition (), mobility->GetVelocity ());
m_neighbors[i].SetTrajectoryBegin (Simulator::Now ());
      // Schedule hello packet sending

      Simulator::Schedule (Seconds (5.0 +i), &AdvertisePosition,
mobility->GetPosition (),           mobility->GetVelocity (),
nodes.Get (i));


      Simulator::Schedule (Seconds (5.0 +i), &AdvertisePositionDegree,
nodes.Get (i),
         m_neighbors[i].CalculateDegree (Simulator::Now () + Seconds
(10.0+i)));


      // Initialize queue for each node
      m_queue[i] = kdtm::Queue (50, Seconds(10));

      // Beacon sink on every nodes
      Ptr<Socket> recv = Socket::CreateSocket (nodes.Get (i), tid);
recv->Bind (InetSocketAddress (nodes.Get (i)
                    ->GetObject<ns3::Ipv4> ()
                    ->GetAddress (1,0)
                     .GetLocal (), beacon_port));

      //NS_LOG_INFO ("Addresse local: " << nodes.Get (i)-
>GetObject<ns3::Ipv4> ()->GetAddress (1,0).GetLocal ());
      //device.allocate
      //device->SetReceiveCallback( MakeCallback( &WaveNetDevice))
recv->SetRecvCallback (MakeCallback (&ReceivePacket));
      //Simulator::ScheduleWithContext (i, Seconds (1.0 + i*0.1),
&GenerateHelloMessage, nodes.Get (i));
   }
        uint32_t
randN;
   for (uint32_t mes = 50; mes < 250; mes+=5)
      {
         randN = rand () % nodes.GetN ();
         Simulator::ScheduleWithContext(randN, Seconds(mes),
&GenerateWarningMessage, nodes.Get (randN));
      }


   // open log file for output
std::ofstream os;    os.open
(logFile.c_str ());
```

```
   // Configure callback for logging
   Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
                    MakeBoundCallback (&CourseChange, &os));

   // Create netanim animation file
   AnimationInterface anim ("animation.xml");

   Simulator::Stop (Seconds (duration));
   Simulator::Run ();
   Simulator::Destroy ();


   // Calculate res
double reach_mean = 0.0;
   std::vector<uint32_t>::iterator recv = m_receivers.begin ();
for (; recv != m_receivers.end (); recv++)
      {
         reach_mean += *recv;
      }
   reach_mean = reach_mean/((double) m_receivers.size ()*nodeNum);
    double rebroad_mean =
0.0;

   for (uint32_t rebr = 0; rebr < m_rebroadcast.size (); rebr++)
{
         if (m_receivers.at(rebr) != 0) {
            NS_LOG_INFO (" rebr: " << rebr
               << " m_rebr " << m_rebroadcast.at (rebr)
<< " m_recv " << m_receivers.at (rebr));
rebroad_mean += m_rebroadcast.at (rebr) / (double)
m_receivers.at (rebr);
         }
}
   rebroad_mean = rebroad_mean/(double) m_rebroadcast.size ();

   NS_LOG_INFO ("\n nb of nodes " << nodeNum
      << "\n Reachability " << reach_mean
      << "\n rebroadcast per covered nodes " << rebroad_mean
      << "\n nb of warning bytes transmitted " << m_warningBytes
      << "\n nb of hello bytes transmitted " << m_helloBytes);

   // close log file
os.close();
    return
0; }
```