Design and Performance Study of Algorithms for Consensus in Sparse, Mobile Ad-hoc Networks

Thesis by

Khaled Alekeish

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Newcastle University
Newcastle Upon Tyne, UK

(Defended November 26, 2010)

Acknowledgements

I am deeply grateful to many people for their help and support during the period of my PhD.

First and foremost, I would like to thank my supervisor, Dr. Paul Ezhilchelvan, for his help, guidance and for the many valuable suggestions he made throughout the work for this thesis. My thesis owes much to the patient supervision Dr. Ezhilchelvan has given.

I would like to acknowledge the financial support of Damascus University that provided the necessary financial support for my PhD.

I also would like to thank Dave Cooper for helping me to organize my results and extend my simulations using TORQUE which helped me a lot to achieve effective simulation.

My thanks go to Einar Vollset for helping me to learn SWANS simulator, and Francois Bonnet for helping in the proof part.

I am grateful to Chris Ritson and Jim Wight for helping to solve the technical problems in the cluster where I used to have my experiments.

Thanks also to the old gang from 10.02: Michele Mazzucco, Osama Younes, Marios Andreou, Christiaan Lamprecht, Joris Slegers, and Chris Smith.

Last but not least, I would like to thank my parents for supporting and encouraging me. Without their constant support I would not have made it.

Abstract

Mobile Ad-hoc Networks (MANETs) are self-organizing wireless networks that consist of mobile wireless devices (nodes). These networks operate without the aid of any form of supporting infrastructure, and thus need the participating nodes to co-operate by forwarding each other's messages. MANETs can be deployed when urgent temporary communications are required or when installing network infrastructure is considered too costly or too slow, for example in environments such as battlefields, crisis management or space exploration.

Consensus is central to several applications including collaborative ones which a MANET can facilitate for mobile users. This thesis solves the consensus problem in a sparse MANET in which a node can at times have no other node in its wireless range and useful end-to-end connectivity between nodes can just be a temporary feature that emerges at arbitrary intervals of time for any given node pair.

Efficient one-to-many dissemination, essential for consensus, now becomes a challenge: enough number of destinations cannot deliver a multicast unless nodes retain the multicast message for exercising opportunistic forwarding. Seeking to keep storage and bandwidth costs low, we propose two protocols. An eventually relinquishing ($\diamondsuit RC$) protocol that does not store messages for long is used for attempting at consensus, and an eventually quiescent ($\diamondsuit QC$) one that stops forwarding messages after a while is used for concluding consensus. Use of $\diamondsuit RC$ protocol poses additional challenges for consensus, when the fraction, $\frac{f}{n}$, of nodes that can crash is: $\frac{1}{4} \leq \frac{f}{n} < \frac{1}{2}$.

Consensus latency and packet overhead are measured through simulation indicating that they are not too high to be feasible in MANETs. They both decrease considerably even for a modest increase in network density.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Mobile Ad-hoc Network (MANET) consists of wireless mobile devices, called *nodes*, which have the ability to send and receive messages. So each node is equipped with wireless transmitter and receiver. These nodes operate without the aid of any fixed infrastructure such as access points. So when a node transmits a message, all nodes within its wireless transmission range may receive this message. In this type of networks, a node wishing to communicate with nodes which are outside its wireless range will have to use intermediate nodes to route messages. So the message might visit multiple nodes before reaching its destination. Such networks are often referred to as *multi-hop* networks.

The number of mobile devices with wireless network capabilities is growing, triggered by the fast growth of wireless communication. The wireless capable devices are daily used by the users of WiFi (802.11X variants) in laptops and PDAs (Personal Digital Assistants), and Bluetooth in mobile phones. The mobility speed of these devices can be wide-ranging; people, vehicles, animals etc may all carry these devices.

MANETs have some constraints which should be considered when designing any solutions for this type of networks. These constraints can be summarized as follows.

1. Dynamic topology: The node mobility causes continuous and unpredictable topology changes of mobile ad hoc networks. The link connectivity dynamically varies in an arbitrary manner. This link is based on the proximity of one node to another node, and it is affected by physical obstacles (such as moving vehicles, buildings, mountains, etc). Thus, the link is prone to frequent breakage during node's mobility.

2. Low bandwidth: MANETs use wireless channels that provide limited band-

width for transmission. Moreover, these channels are subjected to noise, fading and interference.

3. Limited energy: Mobile devices get their energy from batteries which have a limited power. This results in limited processing power of the mobile devices.

1.1 Applications of MANET

Generally, MANETs are suitable for all cases in which a temporary communication is required or in cases the constructing of a traditional network infrastructure is not possible. A common example for the environment where a network infrastructure cannot be created is the military battlefield. Military devices (tanks, vehicles etc) usually include some kind of computer equipment. Ad hoc networking would allow these devices to retain an information network between the military information head quarters, soldiers, and vehicles.

Related heavily to military scenarios are crisis and catastrophe scenario management applications. It is very popular to use ad hoc networks in emergency circumstances. These occur, for example, after a large natural disasters like an earthquake where the entire communications infrastructure may be completely damaged. Reviving communications quickly is crucial. So using ad hoc networks, the communications could be installed in hours instead of using wire-line communications which require days/weeks to be installed.

Another common application of MANET is a Vehicular Ad-Hoc Network ($VANET$). This type of networks provides road safety and passengers comfort. Vehicle moving on the road are equipped with special electronic wireless devices which provide ad hoc network connectivity to the passengers. So each vehicle, supplied with the wireless device, can receive and send/forward messages through the wireless network. The drivers of the connected vehicles can share the road information which includes collision warning, road sign alarms and in-place traffic view which all give the driver essential tools to choose the best path along the way.

1.2 Background and Motivation

MANET is perhaps unique in its ability to facilitate collaboration amongst mobile users in terrains that have no fixed infrastructures for communication support. One

of the challenging problems in supporting collaboration is to enable the users to reach an identical collective decision while their preferred options are all different but equally appropriate. This problem is widely known as the *consensus* [FLP85]. The broader aim of our work is to build a consensus module for supporting collaborations in a MANET.

In the consensus problem, implemented over a group of n nodes, each node N_i proposes a value v_i, then all correct nodes need to decide on a common value v which is equal to one of the proposed values [CT96]. So the consensus problem can be described using two primitives: **propose** and **decide**. Thus, we say that the node N_i "proposes" v_i, where v_i is its proposal to the consensus protocol, when N_i invokes **propose(**v_i**)**. Moreover, we say that N_i "decides" v, when N_i invokes **decide()** and gets v as a result.

The consensus problem is defined by the following properties:

Validity If a node decides v, then v was the proposed value by some node.

Agreement No two nodes decide different values.

Termination Each correct node finally decides.

While the agreement and validity properties define the safety properties associated with consensus, the termination property defines its liveness property.

We consider a system $\mathcal{S}$ made up of N nodes collaborating towards a common goal. Out of N nodes, a small group G of n nodes is formed for the purpose of reaching consensus during the collaboration. So each node in G proposes a value and a unanimous choice out of the different proposals needs to be made. There are several consensus protocols proposed for MANETs. Some of these protocols ([WCR09], [WCYR07], [CDG$^+$05], [BPS08]) require that at least one node in G remains connected to all other nodes in G for a sufficiently long duration. We will derive a consensus protocol depending on different approach which doest not impose this requirement. We will also develop two multicast protocols to be used under the derived consensus protocol. The first multicast protocol will be used to help nodes in G to reach decision. The second protocol will be used by deciding nodes to multicast the decision.

Of the several consensus protocols proposed for a MANET, only a few have been subject to performance evaluation. A closer look at these works reveals that they all assume a network density that would leave the MANET almost always connected. Network density can be defined as the average number of nodes within a disc of

radius equal to the nodes' wireless range. Informally, two nodes are said to be directly connected if they are in each other's wireless range, and simply connected if they are either directly connected or a sequence of one or more directly connected nodes is between in them. (In Figure 1.1, $N_i, 1 \leq i \leq 4$, are connected to each other, where $\leftrightarrow$ is a direct connectivity.) Thus, the denser the network, the more likely that any two nodes remain connected despite node mobility.

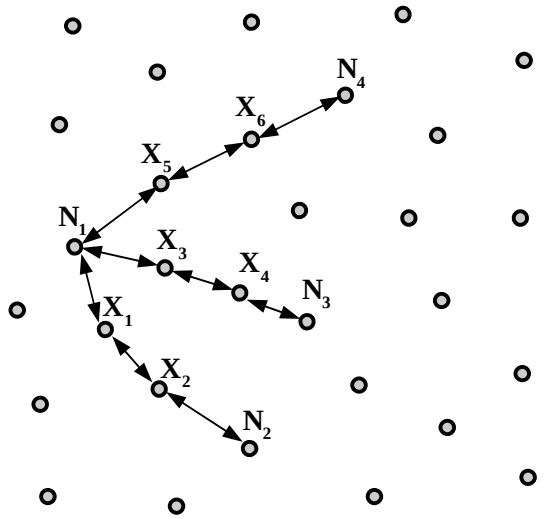

Figure 1.1: Always Connected Network.

Table 1.1 lists the density values considered in the design and evaluation of consensus protocols in the literature. A density value of, say, 8.7 means that each node is expected to be directly connected with 7.7 other nodes at any moment. Even when nodes move at moderate speeds, any two nodes will have, at any moment, at least one path connecting them and also lasting long enough for supporting message transmission. (In [BPS08], connections last longer also for acknowledgments to be received.)

Table 1.1: Network Densities Considered for Evaluation

Protocol	Density
WCR2009[WCR09]	7.8
WCYR2007[WCYR07]	7.8
CDSNT2005[CDG+05]	11.2 - 352
BPS2008[BPS08]	8.7 - 707

At the opposite end to dense networks is the *delay tolerant networking* (DTN)

[Fal03]. Density is so low that a multi-hop connectivity is rare and direct connectivity between a given node pair can take an arbitrarily long time to emerge. So, routing a message, say, m to a given destination itself is a challenge with DTN (see [Zha06] for a survey). It typically relies on opportunistic forwarding as Figure 1.2 illustrates: direct connections that commence at different (and arbitrary) instances, $t_1,\ t_2,\ t_3$ are treated as 'opportunities' for forwarding m from N_1 to N_2, with intermediaries X_1 and X_2 willing to retain m and wait for such an opportunity. Figure 1.2 shows that nodes N_1 and X_1 become connected during the interval (t_1, t_1') in which N_1 transmits m to X_1, node X_1 upon receiving m will retain this message awaiting any opportunity to forward it. After t_1', nodes possessing m do not experience connectivity with any other nodes for an arbitrary time until t_2. During the interval (t_2, t_2'), X_1 and X_2 get connected, so m is transmitted to X_2. In the same way nodes X_2 and N_2 attain connectivity during the interval (t_3, t_3') after an arbitrary disconnectivity interval (t_2', t_3) of nodes which possess m. So nodes in this type of networks manage to exchange messages after a sequence of intermittent connectivity until the message reaches its destination. Message latencies are typically long and are not the only criterion for measuring the efficacy of a routing strategy [Zha06]. To our knowledge, consensus has not been solved with DTN.

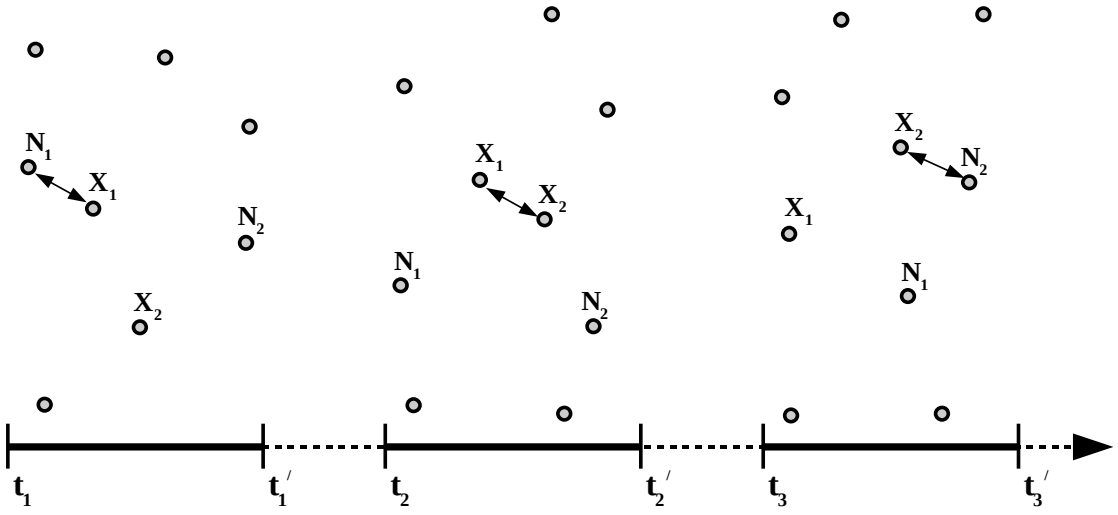

Figure 1.2: Delay Tolerant Network.

This thesis solves consensus for MANETs of density values larger than DTN but not large enough to keep the network always connected. The MANET we consider keeps distinct subsets of nodes connected at distinct intervals, and the sequence of connectivity formed over some (unknown) period of time, would allow any node to

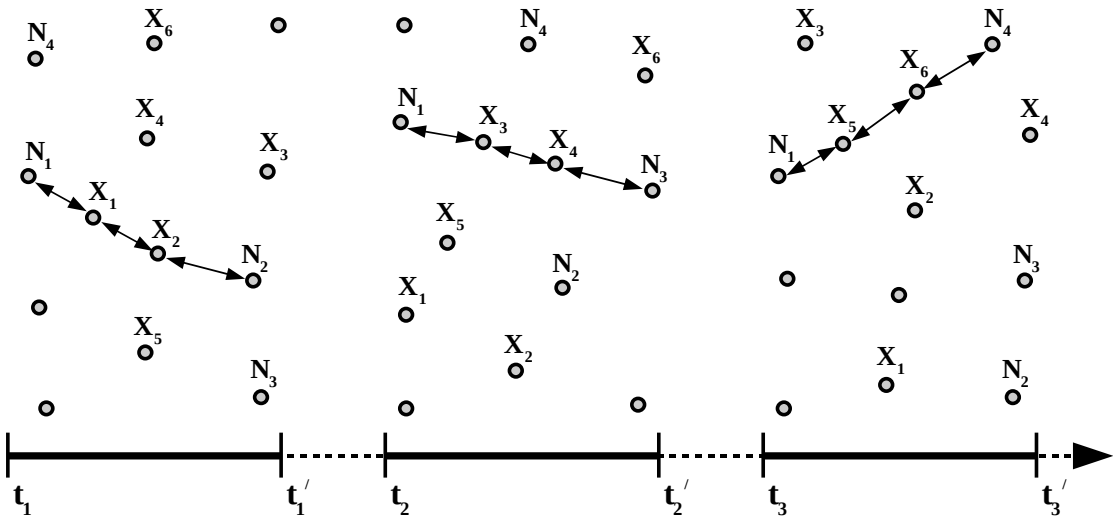

Figure 1.3: A Sparse Network.

transmit its message to any other node if opportunistic forwarding is used.

Referring to Figure 1.3, connectivity involving $\{N_1, X_1, X_2, N_2\}$, prevails during the interval (t_1, t_1'), enables nodes N_1 and N_2 to communicate without the need for intermediate nodes (X_1, X_2) to retain messages. An arbitrary interval of time elapses before the connectivity involving $\{N_1, X_3, X_4, N_3\}$ occurs during the interval (t_2, t_2') and so on. Thus (t_1', t_2) and (t_2', t_3) are arbitrary unknown intervals, but they are finite according to the *liveness* requirement which will be explained shortly. The connectivity sequence occurred during (t_1, t_3') supports opportunistic forwarding from N_2 to N_3 and N_4 (via N_1); and also from N_3 to N_4.

Specifically, we assume that a MANET is just dense enough to satisfy the following *liveness* requirement: for any subset G of three or more nodes, such as $G = \{N_1, N_2, N_3, N_4\}$, any partitioning of G, which is likely to occur, cannot be permanent and is guaranteed to be healed through the connectivity sequence that emerge over some finite but unknown amount of time. When the MANET meets this requirement, nodes of G are guaranteed to be able to exchange information, and therefore collaborate, with each other.

To expose the challenges posed by a sparse network, we simulated a network of 50 mobile nodes dispersed over a square terrain of dimensions $(1000m) \times (1000m)$. The wireless range of nodes is $100m$ resulting in a nodes density of 1.6. So, there is a 40% chance that a node will have no other node in its wireless range at any given moment. 10 nodes were randomly selected to form a group G. The maximum node speeds 5 m/s, 20 m/s, and 40 m/s where nodes move according to the Random

Waypoint mobility style. Nodes of G exchange messages using the basic flooding protocol: when a node receives a message for the first time, it transmits that message after a small random wait (irrespective of whether any node is present in its wireless range).

A flood, once initiated, terminates quickly. The fraction of nodes in G which receive a flood message (at least once), called the delivery ratio (DR), can thus indicate the connectivity that existed between the message source and other nodes of G during flooding. Each node of G flooded 100 messages of 512 Bytes each and, to avoid losses due to collisions, floods (from any node) were well spaced out in time. With node mobility as per Random Way Point model, Figure 1.4 depicts the cumulative distribution function (cdf) of DR. Using this, we present the three significant challenges that need to be addressed for reaching consensus when the MANET is sparse.

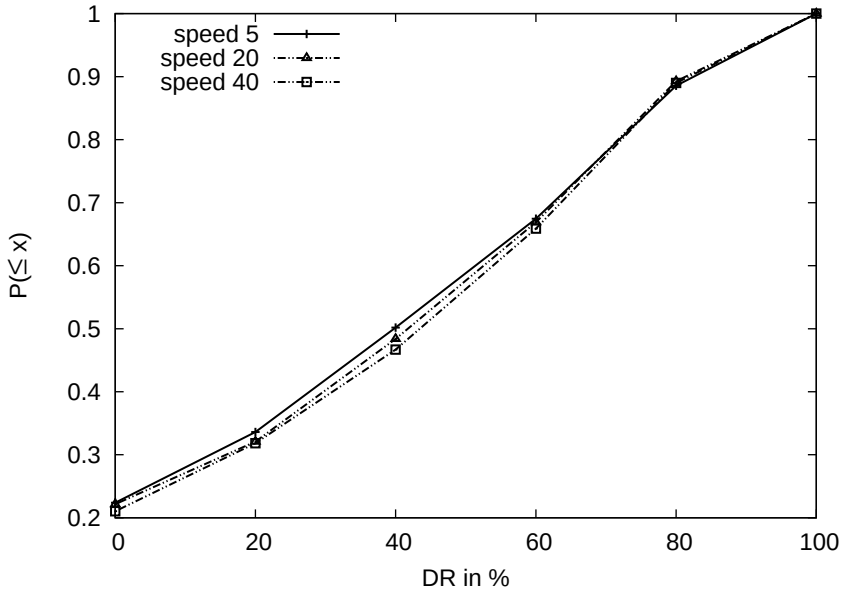

Figure 1.4: CDF of DR. Wireless Range = 100m

- The earlier works listed in Table 1.1 use a class of consensus protocols ([WCR09], [WCYR07], [CDG+05], [BPS08]) that require that at least one node in G remains connected to all other nodes in G for a sufficiently long duration. Figure 1.4 suggests that a sparse MANET cannot be relied upon to meet this requirement: the probability that a node at any moment is connected to, say, more than 80% of nodes is only 10% ($1 - P(DR \leq 0.8) \approx 0.1$). So, we will use a different approach [EMR01] that imposes no such requirement.

- A consensus protocol, randomized or otherwise, requires that when a node multicasts its initial proposal, at least a majority of collaborating nodes receive it. With this probability being as small as 40% in Figure 1.4, consensus requires a multicast support that indulges in opportunistic forwarding which in turn requires nodes to retain messages potentially for long durations. Moreover, nodes typically engage in several rounds of message exchange before reaching consensus and this increases the number of messages being retained.

  So, we develop two types of multicast protocols to support consensus:

  1. ($\diamondsuit RC$) protocol that is guaranteed to delete a message (eventually relinquishing $\diamondsuit R$) once a target on DR is achieved (*coverage*, C). This protocol will be used to help nodes to reach decision.

  2. An eventually quiescent ($\diamondsuit QC$) protocol that is guaranteed not to transmit a message m (eventually quiescent $\diamondsuit Q$) once all nodes that were operative at some time have received m (*coverage*, C), but may retain m for ever. So $\diamondsuit QC$ will be used just for sending the decision when a node decides. This protocol, unlike $\diamondsuit RC$, guarantees that each operative node will receive the decision if it does not decide, otherwise the decisions will be identical.

- The third and final challenge arises due to an interplay between the possibility of node crashes and the maximum DR the $\diamondsuit RC$ protocol can guarantee. Even if the latter delivers a multicast message to a majority of nodes, some of these nodes could subsequently crash, leaving the number of operative nodes that received the multicast to fall below the threshold needed for consensus to make progress. To deal with such unfortunate possibilities, the consensus protocol of [EMR01] will have to be appropriately adapted.

When a node decides, the decision should be sent to all operative nodes. So the $\diamondsuit QC$ protocol will be used to send decisions. The $\diamondsuit Q$ protocols in [ACT00] assume that the nodes are connected. The latter assumption cannot be tolerated in sparse MANETs. So the $\diamondsuit QC$ protocol will be developed to suit sparse MANETs. Moreover, another reason for using $\diamondsuit QC$ only for sending decisions is that: $\diamondsuit QC$ might retain messages forever. The authors of [ACT00] also observe that for $\diamondsuit Q$ protocol to be able to delete messages (have $\diamondsuit R$), nodes need to know which nodes

have crashed. The latter knowledge is very difficult to obtain in MANETs, it is particularly difficult to distinguish slow nodes from crashed ones.

$\diamondsuit RC$ protocol, unlike $\diamondsuit QC$, is eventually relinquishing ($\diamondsuit R$) which means that it deletes each message m after meeting a target on DR. The latter $\diamondsuit R$ property is of high interest in MANETs in which nodes suffer from memory constraints. Therefore, the use of $\diamondsuit RC$ protocol will be kept to maximum possible. Basic flooding and its variations, such as [HOTV99], are $\diamondsuit R$ protocols. However, they cannot guarantee anything on the DR, this guarantee is a crucial requirement for reaching consensus.

$\diamondsuit RC$ and $\diamondsuit QC$ protocols described here use two, simple underlying services.

1. The first service provides, for a node N_i, an up-to-date view on the H-hop neighborhood $Neigh_H$, which contains all other nodes that are connected to N_i by at most H hops, for some chosen $H \geq 1$.

2. The second service is optional and it can be any $\diamondsuit R$ (eventually relinquishing) protocol such as basic flooding. Using this service reduces the message overhead of $\diamondsuit RC$ and $\diamondsuit QC$ multicast messages, albeit at the expense of delivery latency. Its presence offers a leverage for making a trade-off between overhead and latency.

1.3 Contributions

The main contributions of this thesis can be summarised as follows:

1. $\diamondsuit RC$ **Multicast Protocol**
   a new $\diamondsuit RC$ multicast protocol is developed to support the consensus. This protocol guarantees that nodes discard every multicast message m once a target on the number of nodes which have received m is achieved (Chapter 5).

2. $\diamondsuit QC$ **Multicast Protocol**
   We introduce a new $\diamondsuit QC$ multicast protocol which guarantees that dissemination of a multicast message m stops permanently once all operative nodes have received m (Chapter 5).

3. **Derivation of A Consensus Protocol**
   We derive a consensus protocol which forms with the latter two multicast pro-

tocols a consensus module for supporting collaborations in a MANET (Chapter 6).

4. $\diamondsuit R$ **Multicast Protocol**
   $\diamondsuit QC$ and $\diamondsuit RC$ protocols can work on top of any $\diamondsuit R$ multicast protocol. So we produce a new $\diamondsuit R$ multicast protocol to be used by our $\diamondsuit QC$ and $\diamondsuit RC$. This protocol is derived from an encounter based broadcast protocol (Chapter 4).

5. **Neighbourhood Manager *NM* Protocol**
   We develop a new protocol, called Neighbourhood Manager *NM*, to build and maintain the H-hop neighbourhood list. This list is used by $\diamondsuit QC$ and $\diamondsuit RC$ protocols (Chapter 5 Section 5.4).

1.4 Thesis Outline

The outline of the thesis is as follows. In Chapter 2 we discuss the related work for both consensus and multicast protocols. We highlight the limitations and merits of the various approaches.

In Chapter 3 we describe the system context, fault assumptions which define the maximum number of tolerable crashes in the system, required connectivity assumptions for the system to be able to reach consensus, specifications of multicast protocols and our approach to reaching consensus.

In chapter 4 a $\diamondsuit R$ protocol called Encounter Gossip Multicast protocol is derived from another protocol (Encounter Gossip broadcast protocol), and a performance evaluation of the new derived protocol is carried out. Moreover, we introduce an optimization to reduce the number of redundant floods in our new $\diamondsuit R$ protocol and study the performance after implementing the new optimization.

In chapter 5 we introduce our new $\diamondsuit RC$ and $\diamondsuit QC$ multicast protocols on top of the Encounter Gossip Multicast protocol, present the new Neighbourhood Manager protocol and study the performance of $\diamondsuit RC$ and $\diamondsuit QC$ protocols.

In chapter 6, we present our derived consensus protocol and study the performance of the whole consensus module, that is, the consensus protocol on top of $\diamondsuit RC$ and $\diamondsuit QC$ protocols.

In Chapter 7, we conclude by summarizing our contributions and giving some outlines for the future work.

Chapter 2

Related Work

This chapter discusses fundamental related work on multicast routing protocols and solving the consensus problem in MANETs. We give a brief view about different types of multicast routing protocols for MANETs. The multicast routing in Delay Tolerant Networks (DTN) needs more requirements, in terms of memory and power, so it is briefly discussed in a separate section (in section 2.1). The taxonomy of multicast routing protocols in dense networks is introduced in section 2.2. In section 2.3, detailed discussion about the consensus problem is introduced. This includes the known approaches and some existed protocols for solving consensus.

2.1 Routing in Delay Tolerant Network (DTN)

The routing of *DTN* can be categorized into two types deterministic routing and stochastic (dynamic) routing [Zha06].

2.1.1 Deterministic Routing

This type of routing assumes that the future movement of nodes is completely known. So all the future connections between nodes are known ahead of time. Therefore, the transmissions of any message m (when and where to forward m) can be scheduled ahead of time. So an end-to-end route is determined before a message is actually transmitted. We refer the reader to [HGR04] for more information about deterministic routing.

The deterministic routing assumes that the network topology is known ahead of time. This assumption cannot be tolerated in MANETs where nodes move randomly

and in unpredictable way. In particular, in environments such as a battle field or a city hit by earthquake.

2.1.2 Stochastic Approach

Protocols under this approach assume that the network behavior is random and not known. These protocols have to make decisions regarding when and where to forward messages. The easiest decision is to transmit to any contacts within range, while other ways are based on mobility patterns or history data.

Xi *et al.* [XC09] propose an encounter-based multicast routing (*EBMR*) protocol for *DTN*. This protocol depends on the knowledge, achieved from the history of encounters, to select a next-hop node for sending a message m. Specifically, this protocol creates, at every node N_i, a probabilistic variable called delivery predictability for each known destination. This variable shows the probable ability of node N_i to deliver a message to the destination N_j. Moreover, nodes exchange their delivery predictability values by broadcasting beacon periodically. So the beacon, which is sent by N_i, contains the delivery predictability values from N_i to all other nodes. When node N_j receives the beacon, it updates its delivery predictability values according to the received ones. Moreover, the delivery predictability for node N_i will age if N_j does not receive beacon from N_i during a specific interval of time. The delivery predictability is transitive. So when N_j encounters N_k and N_i encounters N_j, N_i can update its delivery predictability to node N_k (denoted as $P(N_i, N_k)$) depending on its delivery predictability to node N_j $P(N_i, N_j)$ and node N_j's delivery predictability to node N_k $P(N_j, N_k)$.

The delivery predictability values are maintained in an $N \times N$ Matrix (N is the total number of nodes) where row i in the matrix contains the delivery predictability values of node N_i to the other $N - 1$ nodes (the diagonal entry (i, i) is ignored).

According to *EBMR*, node N_i, which encounters another node N_j, passes the multicast message m to N_j if N_j has delivery predictability higher than a delivery threshold (P_{thresh}) or a wait timer (WT) expires. When a node receives a multicast message, it will select the required number of nodes with the highest delivery predictability (that exceeds P_{thresh}) to every multicast receiver. The message will be stored if no such next-hop node is found, that is, until the wait timer WT expires. When WT expires, the node will select a node, which has not been selected before,

with the highest delivery predictability to a multicast receiver. So each relay node (including the source) will have the chance to pick a next-hop node with delivery predictability greater than P_{thresh} within the WT time.

Xi *et al.* state that the use of WT, for nodes storing a message m, reduces the number of hops which are visited by m before reaching the destination. That is, by giving the node N_i, which stores m, the chance to continue to store m during the WT time. So N_i can have longer time to choose a next hop, for m, which has delivery predictability higher than P_{thresh}. However, in cases when nodes need to engage in several rounds of message exchange (such as consensus), they are required to retain more messages at one time. This might result in unaffordable cost of memory and power by the nodes which suffer from memory and power constraints. Moreover, any node which has a message m deletes m upon choosing the next hop for each receiver. This means that the channel is assumed to be reliable which is not convenient assumption for MANETs where m might get lost after being transmitted.

The analysis and simulations, which the authors had for this protocol, show that the average end-to-end delay incurred by the delivered messages was about 2000 seconds. That is, by using the Random Waypoint mobility style, $P_{thresh} = 0.5$ and $WT = 500$ seconds. The authors also show that this delay can be reduced when the source of each sent message m transmits m more than once. However, it is shown that this reduction is at the expense of data efficiency (the total number of messages received to the number of transmissions used to deliver these messages) which is reduced dramatically when a source transmits a message more than once.

2.2 Taxonomy of Multicast Routing Protocols in Dense Networks

Classification methods assist designers and researchers to comprehend the distinct features of different multicast routing protocols and discover the internal relationship among them. These features are mainly related to the tasks which nodes may take in the multicast routing process and the information which is exploited for MANETs. We consider below two types of classifications.

2.2.1 Tree and Mesh Based Multicast Routing Protocols

One of the most common ways of classifying multicast protocols is based on how routes among group members are built. According to this way, multicast protocols are divided into mesh-based and tree-based multicast protocols.

Many performance comparison studies of tree-based and mesh-based multicast protocols were carried out. In this section, we will refer to two of these studies [LSH+00] and [VOT06].

The study in [LSH+00] compares between the performance of On-Demand Multicast Routing Protocol (*ODMRP*) [LGC99] and Core-Assisted Mesh Protocol (*CAMP*) [GLAM99] as mesh-based approaches versus Adhoc Multicasting Routing Protocol (*AMRoute*) [BLMT98] and Ad Hoc Multicast Routing Protocol Utilizing Increasing Id-NumberS (*AMRIS*) [WTa98] which are both tree-based protocols. This study shows that *ODMRP* has the best performance against all other protocols included in this study. Moreover, *CAMP* was the second best protocol which also gives a good performance comparing with the tree-based approaches.

The another study in [VOT06] compares *ODMRP* against Multicast Ad Hoc On-Demand Distance Vector (*MAODV*) protocol [RP99] which is a tree-based protocol. This study proves that *ODMRP* gives a better packet delivery ratio than *MAODV*.

The mesh-based protocols can provide with a high delivery ratio because they create redundant routes with a mesh topology. This enables these protocols to work even when some routes break due to node mobility. Therefore, mesh-based protocols can be described as robust even when the node mobility is high. In contrast, tree-based protocols give poor delivery ratio when node mobility is high. In fact, the absence of redundant routes compels these protocols to rebuild the routes tree when any topology changes occur. Therefore, these protocols are considered as very fragile to nodes mobility.

2.2.2 Proactive and Reactive Multicast Routing Protocols

Another taxonomy is based on how routing information is gathered and maintained by mobile nodes. According to this method, multicast protocols can be classified into proactive routing and reactive routing.

A proactive multicast routing protocol is denoted as “table-driven” multicast protocol. According to this type of protocols, each node retains one or more tables

containing information about the entire network topology. These tables retain up-to-date routing information from every node to each other node. Maintaining up-to-date routing information requires nodes to exchange the topology information on a regular basis. This leads to a relatively excessive overhead on the network. However, routes between nodes will always be available on request. Recently, several proactive multicast routing protocols have been proposed, such as ad-hoc multicast routing protocol utilizing increasing Id numbers (AMRIS) [WTa98], core assisted mesh protocol (CAMP) [GLAM99] , and location guided tree (LGT) [CTA02].

A reactive multicast routing protocol is denoted as "on-demand" multicast protocol. Reactive protocols build the routes on-demand. That is, just when a node needs to commence communication with another node to which it has no route, the protocol will attempt to set such a route. Reactive multicast protocols are more scalable than proactive multicast protocols because they build routes just when these routes are needed. However, when using reactive multicast protocols, source nodes may need to wait until the routes are established. So source nodes may practice long delays for route searching before they can send data packets. As examples for reactive multicast protocols, Adaptive Core-based Multicast Routing Protocol (ACMP) [KEJ05] and Mesh-based Multicast Routing Protocol with Consolidated Query Packets (CQMP) [DN05].

2.3 The Consensus Problem

2.3.1 System Model

One of the most essential characteristics of distributed systems is the distinction between the *synchronous* and the *asynchronous* systems. In the *synchronous* system each message can be received in a known bounded time after it is sent, also any process take a known bounded time to be performed. While in the *asynchronous* system the bounds on the message delays and the processing speeds are not certain. This makes the asynchronous systems a very generic paradigm for distributed systems. It is clear that MANETs are actually asynchronous systems because in MANETs the bounds on message-delivery are not known with certenity. In particular, when any two nodes get disconnected, it cannot be known how long it will take for these nodes to get connected again.

2.3.2 A Fundamental Impossibility Result

This impossibility has been proved by Fischer, Lynch and Paterson[FLP85]. They showed that it is impossible to design a deterministic consensus protocol in an asynchronous distributed system prone to even a single crash failure. This result is known as the *FLP impossibility result* and it holds even if a reliable communication between nodes is attained.

The grounds which lead to this impossibility are that in an asynchronous system it is impossible to recognize a crashed node from a very slow one. So the nodes might wait to receive from one node (which is an expected slow node), but this node may crash without the knowledge of other nodes. As a result, the nodes will wait for ever to receive from this crashed node and so they will never decide. This contradicts the termination property in the definition of consensus.

System developers must have a deep understanding about this impossibility result [GS97] which has attracted many researchers to find the weakest conditions which, when satisfied by asynchronous systems, cause the consensus problem to be solvable. The next section will survey some approaches used to solve the consensus problem.

2.3.3 Known Approaches to Solving consensus

Many protocols have been developed to circumvent the *FLP impossibility result*. These protocols use two ways in their solutions; (i) adding some timeliness requirements to the network, or (ii) changing the deterministic termination property to a probabilistic one.

The first way, adding some timeliness requirements, assumes that the network moves through steady intervals during which the progression towards consensus can be achieved. This way involves the *partial synchrony* [DLS88], the *timed asynchronous* model [CF99] and *unreliable failure detectors* [CT96]. Due to the popularity of failure detectors, we will elaborate on them shortly.

The second way, changing the deterministic termination property, solves the problem by weakening the deterministic termination property to the termination with probability 1. That is, without affecting the safety properties. The *randomization* [BO83] uses this way to solve the problem.

2.3.3.1 Unreliable Failure Detectors

The unreliable failure detector concept, which was first established by Chandra and Toueg [CT96], is one of the strongest notions for creating and maintaining dependable distributed applications. So, in a system with n nodes, the failure detector consists of n modules, one per node, which provide the nodes with approximate views of the node crashes during the execution. Thus each module provides a list of suspected/crashed nodes. Since the system is asynchronous, the failure detector is expected to make mistakes. That is, a failure detector may not suspect a crashed node, or suspect a correct one. However, the failure detector eventually has to give true information. So the errors of any failure detector has to be constrained by *completeness* and *accuracy* properties. The completeness property needs nodes to eventually suspect every bad node, while accuracy confines the number of incorrect suspicions which a failure detector might make.

Several classes of failure detectors were defined in [CT96] based on different descriptions of the completeness and accuracy properties. In this section we will concentrate on the class of *eventual strong* failure detectors, denoted as $\diamondsuit S$, which are the weakest required for consensus [CT96]. The completeness and accuracy properties for $\diamondsuit S$ class are defined as follows:

Strong Completeness: Eventually all correct nodes permanently suspect every crashed node;

Eventual Weak Accuracy: Eventually some correct node is never suspected by any correct node;

We will also define the class of *eventual perfect* failure detectors ($\diamondsuit P$). This class of failure detectors has the same completeness property of $\diamondsuit S$ and the following accuracy property:

Eventual Strong Accuracy: There is a time after which correct nodes are not suspected by any correct node.

The failure detectors build the list of suspected nodes by following one of two ways; either by sending periodic "heartbeat" messages between the failure detector modules or by depending on gossiping to exchange the view of suspected nodes [vRMH98].

2.3.4 $\diamondsuit R$ and $\diamondsuit Q$ protocols

The basic flooding protocol [HOTV99] is a $\diamondsuit R$ protocol. This protocol works in very simple way; when a node has a message m to send, it transmits m for one time. Each node upon receiving m for the first time, it transmits m after a small random wait (irrespective of whether any node is present in its wireless range).

The flooding protocol is network topology-independent which means that it does not use any information about the network topology. The latter property is attractive for MANETs because nodes move frequently so storing valid information about the network topology becomes a big challenge. However, the flooding protocol cannot guarantee anything on the number of nodes that would receive a sent message m. This guarantee is an important requirement for many applications (such as consensus).

The authors in [ACT00] propose their quiescent reliable communication protocols, and they have concentrated on two categories of reliable communication mechanisms: *reliable broadcast* and *quasi-reliable send and receive*. Reliable broadcast guarantees that (i) when a correct node (e.g., it does not crash) broadcasts a message m, all correct nodes receive m, and (ii) all correct nodes receive the same group of messages. The primitives of send and receive are said to be quasi-reliable if they meet the condition: when nodes p and q are correct, q will receive a message m from p precisely as many times as p sent m.

The authors of [ACT00] have proved that there is no quiescent implementation of reliable broadcast or quasi-reliable send and receive in asynchronous systems subject to at least one crash (with no failure detectors). For example, if a node p is sending a message m and a neighbour q, of p, crashes before receiving m, p will keep sending m to q for ever because p does not know that q is crashed. So the authors use a failure detector to achieve a quiescent implementation. The failure detector used here is the *Heartbeat* (*HB*) failure detector. *HB* works in a simple way and it requires each node p to store a vector of counters (one counter for each neighbour q of p). So this detector works according to the following steps: (i) each node periodically sends a small message called a 'heartbeat', and (ii) when a neighbour q of p receives a heartbeat from p, it increases the stored counter for p. So if a neighbour p of q is not crashed, its counter at q increases with no bound. Therefore, when p crashes, its counter at a neighbour q finally stops increasing.

So the authors of [ACT00] developed a number of algorithms for performing a quiescent communication using HB. We choose to describe the implementation of reliable broadcast because this implementation is more relevant to our approach. In this implementation, each node p stores, for each message m, a knowledge vector $has_p[m]$ containing a set of nodes. A node q is added to $has_p[m]$ if p knows that q has received m. Each sent message m by a node p is in the form (m, has_msg, $route$) where has_msg is the current value of $has_p[m]$, and $route$ is the list of nodes that this copy of (m, has_msg, $route$) has visited so far.

For each broadcast message m, p initializes $has_p[m]$ to $\{p\}$ and calls the task $diffuse(m)$. In the latter task, which runs in the background, node p periodically looks at its HB; if, for a neighbour $q \notin has_p[m]$, the counter of q at p has grown p sends (m, $has_p[m]$, p) to all neighbours whose counter has grown. $diffuse(m)$ runs until all neighbours of p are included in $has_p[m]$.

When a node p receives a message (m, has_msg, $route$): (i) if it has not already received m, it initializes $has_p[m]$ to $\{p\}$ and calls the task $diffuse(m)$, (ii) it annexes the content of the received has_msg to $has_p[m]$ and appends itself to $route$, and (iii) it sends the new message (m, $has_p[m]$, p) to the neighbours which show up at most once in $route$.

2.3.5 Some Existing Protocols for Solving Consensus

Lamport [Lam06] proposes a consensus protocol denoted as Fast Paxos. According to Fast Paxos, a single node can play different roles by enacting different agents at the same time. An agent can represent one of the following major roles: a proposer which can propose values to send them to acceptors, an acceptor that participates in choosing a single value to be sent to learners, and a learner which receives, and learns, the value that has been chosen.

Fast Paxos proceeds in rounds, with each round having two phases. This protocol also uses a group of nodes which can act as coordinators; Any acceptor can play the role of coordinator for some round. Moreover, a coordinator might coordinate a lot of rounds. A coordinator c retains the following information:

$crnd[c]$ The greatest round number which c has initiated, initially 0.

$cval[c]$ The value that was chosen by c in the round $crnd[c]$, or $none$ when no values

have been chosen by c.

Each acceptor a retains the following information:

$rnd[a]$ The greatest round number in which a has played a role, initially 0.

$vrnd[a]$ The largest round number in which a has diffused a vote, initially 0.

$vval[a]$ The value which was accepted by a in round $vrnd[a]$.

A round r, which is coordinated by c, proceeds in two phases according to the following steps:

Phase 1:

a) A coordinator c, which changes $crnd[c]$ to r and $cval[c]$ to none, tries to request a participation in the round r from each acceptor a by sending a message to every acceptor a

b) When an acceptor a receives a message that asks a to participate in round r and $r > rnd[a]$, it changes $rnd[a]$ to r and replies to the coordinator c by sending a message that includes the round number r and the stored values of $vrnd[a]$ and $vval[a]$

Phase 2:

a) If a coordinator c receives the replies (which was sent in step b in phase 1) from the majority of acceptors in a round r, $crnd[c] = r$ and $cval[c]$=none; Then the coordinator c uses the received replies to choose a value v; where c chooses the reply that contains the greatest $vrnd[a]$ and copies $vval[a]$ from that reply to v. Thereafter, c changes its $cval[c]$ to v and sends a message to all acceptors to ask them to vote in round r to accept v

b) An acceptor a upon receiving a requesting message to vote on a value v in round r, it will accept v if $r \geq rnd[a]$ and $vrnd[a] \neq r$. After accepting v, a sets $vval[a]$ to v, sets $vrnd[a]$ and $rnd[a]$ to r, and then tries to inform all learners about its vote in round r by sending a message to these learners.

Wu *et al.* [WCYR07] propose a consensus protocol equipped with $\diamondsuit P$ which is unreliable failure detector FD [CT96], [LFA04]. The FD provides nodes with the list of nodes which are suspected to crash or have crashed. The proposed consensus protocol divides the network into two layers; The first one represents the Clusterhead layer which contains a predefined group H of nodes that function as clusterheads to merge/unmerge and forward messages for nodes. The second one is the Host layer which contains a group M of all nodes including those nodes in H.

Only nodes in H behave as coordinators or decision makers/agreement keepers

DA. So, each node links itself to the nearest unsuspected clusterhead from H. Moreover, when a node suspects its clusterhead or it changes position, it has to associate itself with the nearest clusterhead from H.

According to this protocol, each round r is composed of two phases. At the beginning of the first phase of round r, the current coordinator m_{cc} sends its estimate ($PROP(r, est_{cc})$ message) to the nodes in H. Each clusterhead, upon receiving from m_{cc}, forwards the $PROP$ message to its local nodes. In case a clusterhead adds m_{cc} to its suspected nodes before receiving $PROP(r, est_{cc})$, it sends $PROP(r, \perp)$ to its local nodes, where $\perp$ is a value which cannot be adopted and it is different from any estimates of nodes. A node m_i upon receiving the $PROP$ message from its local clusterhead, if the received $PROP(r, v)$ has $v \neq \perp$, then m_i changes its estimate to v and timestamp ts to r. When m_i suspects its local clusterhead or this clusterhead is no longer the nearest, it has to associate itself with another clusterhead before continuing with the consensus protocol.

In the second phase, each node m_i in M sends an echo message $ECHOL(r_i, est_i, ts_i)$ to its local clusterhead. Every clusterhead, after receiving $ECHOL(r, -, -)$ from all unsuspected local nodes, creates its echo message $ECHOG(r, v, ts_v, x, y)$ by combining the received $ECHOL(r, -, -)$ messages. Where ts_v is the highest received timestamp, v is the estimate received in $ECHOL(r, -, -)$ with ts_v, x is the group of nodes that send $ECHOL(r, -, -)$ with ts_v and y is the group of nodes that send $ECHOL(r, -, -)$ with $ts < ts_v$. The created $ECHOG(r, v, ts_v, x, y)$ is then sent to the nodes in DA. Each clusterhead in DA waits until i) receiving $ECHOG(r, -, -, -, -)$ messages which represent no less $n - f$ nodes (all nodes in x and y not less $n - f$), or ii) receiving $ECHOG(-, -, ts_v, -, -)$ message with $ts_v > r$. A clusterhead changes its estimate to the value v which is received with the highest timestamp. If a clusterhead in DA has $f + 1$ or more nodes in x groups of the received $ECHOG(r, v, ts_v, x, y)$ messages with $ts_v = r$, then it decides upon v and broadcasts this decided value.

Wu *et al.* address the problem caused by the reliable channel assumption, but they note that complex design changes would be required to enable their protocol to work with lossy channels, which is more feasible for MANETs. Moreover, this protocol imposes two-layer hierarchy where each node is associated with a clusterhead. This requires nodes, due to mobility and clusterheads crashes, to change their clusterheads during execution. So choosing and agreeing on clusterheads involve

solving consensus which results in circularity. In addition to that, keeping nodes in clusters is almost impossible when the network is sparse.

Wu *et al.* propose another consensus protocol [WCR09] which is similar to their protocol in [WCYR07] in the sense that it makes use of the cluster-based hierarchy. However, Wu *et al.* address three improvements of [WCR09] over [WCYR07]; Firstly, in [WCR09] the clustering function is separated from achieving consensus while this function is tightly coupled with achieving consensus in [WCYR07]. Secondly, the group of clusterheads is changed to be dynamic in [WCR09], which is static in [WCYR07]. Finally, the protocol in [WCR09] is provided with the commonly used FD of $\diamondsuit S$ [CT96] while the $\diamondsuit P$ is used in [WCYR07].

Wu *et al.* [WCR09] separate the concerns of constructing the hierarchy and achieving consensus by specifying the function of clustering nodes as a new oracle, called *eventual clusterer*, denoted as $\diamondsuit C$, and introducing a protocol for achieving consensus using $\diamondsuit C$. The oracle $\diamondsuit C$ has two tasks; detecting the crashes of nodes (based on the FD of $\diamondsuit S$) and building a cluster-based two layer hierarchy. The second task can be further divided into two subtasks: 1)choosing clusterheads and 2)creation and maintenance of clusters.

The selection of clusterheads involves consulting $\diamondsuit S$ about the suspected nodes. Thus, each node excludes any suspected nodes, by its FD, from its list of clusterheads CH. A node m_i periodically sends its CH to other nodes, so these nodes, upon receiving from m_i, exclude any clusterheads which do not appear in the received CH.

The clustering operation is started by cluster members. A node m_i chooses the nearest node m_{cc} from CH and sends a *JOIN* message to m_{cc}. Upon receiving the *JOIN* message, in case of m_{cc} is a clusterhead of itself, it allows the joining of m_i by sending a positive *ACK* message; otherwise, it does not allow the joining of m_i by sending a negative *ACK* message. If a negative *ACK* is received, m_i picks another candidate from CH and repeats the joining procedure; otherwise, it ends this joining procedure.

A clusterhead m_{cc} might get deleted from CH. When a cluster member m_i detects the deletion of the local clusterhead, or receives a *RELEASE* message from the local clusterhead, it will switch to a new cluster. Moreover, a node m_i will switch its cluster, if it discovers that its current clusterhead is no longer the nearest one. If m_i is switching its clusterhead it sends a *LEAVE* message to its current

clusterhead. When m_i itself is the clusterhead being deleted from CH, it sends a *RELEASE* message to tell its cluster members.

Wu *et al.* tried to improve their previous protocol [WCYR07] by introducing the protocol in [WCR09]. The new protocol mainly separates the concerns of constructing the hierarchy and achieving consensus. However, the new protocol still assumes that the channel is reliable. Moreover, building the two-layer hierarchy is not attainable when the network is sparse.

Borran *et al.* [BPS08] propose a consensus protocol denoted as *LastVoting*. *LastVoting* is composed of a sequence of phases, where each phase consists of 4 rounds. In each phase processes have to elect and maintain a coordinator which might be different from phase to phase. Each process has its proposal x_p which is attached with a timestamp ts_p. During the first round of each phase, (x_p, ts_p) is sent by each process p to its coordinator. When the coordinator receives (x, ts) from a majority of processes, it changes its vote to the received proposal with the highest timestamp. In the second round, the vote is sent by the coordinator to all processes. Upon receiving the vote from the coordinator, each process replaces its proposal, by the received vote, and updates its timestamp. In the third round, each process, which has received the coordinator's vote, sends an *ack* message to the coordinator. The coordinator can decide after receiving *ack*s from the majority of processes. In the last round, the coordinator sends the decision to all processes. Each process decides upon receiving the decision from the coordinator.

The coordinator communicates with other processes through diffusion; it broadcasts the message to other processes inside its wireless range. A process p, upon receiving a message from another process q for the first time, becomes a child of q only when $priority_q > priority_p$ (q wins the election against p). Then p, except when $priority_q < priority_p$, broadcasts the received message. Multiple copies of the same message are ignored. The nodes respond to the coordinator by using convergecast which employs the tree which was built during the diffusion.

LastVoting protocol solves the consensus problem in presence of crashes and lossy channels. However, the nodes have to elect their coordinator which, due to crashes and mobility, need to be synchronized periodically. So when nodes get partitioned, they might choose different coordinators. This makes nodes take longer time and more phases to decide. Moreover, synchronizing the coordinator between nodes is very challenging when the network is sparse. In addition to that, the tree of routes

which is used in the convergecast needs to be maintained against mobility.

Chockler *et al.* [CDG+05] developed their consensus algorithms. According to these algorithms, the network consists of a sequence of non-overlapping grid squares where every square is supposed to be populated. Each node has the knowledge of its approximate position in the grid square. A single-hop consensus algorithm first runs within each grid square to reach a local decision which is then sent to all other grid squares. As soon as a node receives a value from each grid square, it decides by applying a deterministic function to the set of received values.

Chockler *et al.* had very complex assumptions on the system model. For example, they put strong synchrony assumptions (inter-node communication delay are bounded by known constants). Moreover, it was assumed that nodes know their locations and they do not move. All these assumptions add more restrictions which cannot be met in MANETs.

2.4 Summary

The routing protocols proposed for DTN suffer from many drawbacks: some of these protocols assume that the network topology is always known, some others assume that links are reliable. These assumptions cannot be tolerated in MANETs where nodes move in unpredictable way and the links are lossy. Moreover, these protocols require nodes to retain messages for opportunistic forwarding. This causes unffordable cost of memory and power in particular when nodes need to engage in several rounds of message exchange.

The multicast routing protocols for MANETs can be categorized as tree/mesh based protocols and proactive/reactive protocols. It was proved that mesh based protocols perform better than tree based protocols in terms of the delivery ratio because mesh based protocols create redundant routes with a mesh topology. This enables the latter protocols to work even when some routes break due to node mobility. In contrast, tree based protocols need to rebuild the routes tree when any topology changes occur.

Reactive protocols are more scalable than proactive protocols because they build routes just when these routes are needed. In contrast, building and maintaining routes in proactive protocols continue for the whole protocol lifetime. This requires nodes, for the whole protocol lifetime, to exchange messages to maintain up to date

routes.

The *FLP impossibility result* proves that it is impossible to design a deterministic consensus protocol in an asynchronous distributed system prone to even a single crash failure. So several protocols were developed to circumvent this impossibility. Of the several developed protocols, only a few have been subject to performance evaluation. So we choose to explain about these protocols and show their weakness in solving the consensus, especially when the network is sparse.

Chapter 3

System Model and the Approach

3.1 System Model

The system $\mathcal{S}$ is made up of mobile nodes collaborating towards a common goal in a terrain that has no fixed infrastructure for supporting communication between nodes. The nodes communicate using the omnidirectional wireless transmission functionality of a CSMA/CA-like MAC layer protocol (e.g. IEEE 802.11b). Exchange of information between nodes is thus limited strictly to ad-hoc networking.

A small group G of n nodes is formed at time t_0 for the purpose of reaching consensus whenever a unanimous choice out of different possibilities needs to be made during the collaboration. Nodes of ($\mathcal{S}$-G) cooperate to discover and maintain connectivity between nodes of G; that is, nodes of ($\mathcal{S}$-G) act as routers for nodes of G to exchange messages and execute consensus.

We assume that nodes of $\mathcal{S}$ have unique identifiers and let G=$\{N_1, N_2, ...N_n\}$ and $\mathcal{S}$-G=$\{X_1, X_2, ...\}$. We also assume that $n \geq 3$ and $|\mathcal{S}\text{-}G| \gg n$. Nodes of G are referred to as member nodes or simply as *members*, and those of $\mathcal{S}$-G as non-member nodes or *non-members*, for short.

Of the n members of G, at most f, $0 < f < \frac{n}{2}$, can crash over the lifetime of G and a crashed member does not recover. Thus, a member is either *working* or *crashed permanently*. A working member is also referred to as an *operative* member and functions according to its specification. Every operative member knows the identity of all other members in G and also knows f and n.

We define $W(t)$ as the set of members that are operative at time $t > t_0$. Since a crashed member does not recover, $W(t)$ is non-increasing over time: $W(t') \subseteq W(t)$,

$\forall t' \geq t$. Since $f < \frac{n}{2} < n$, G has at least $(n - f)$ members that never crash. These members are called *permanently working* members or simply the *correct* members of G. PW denotes the set of all correct members. $PW \subseteq W(t)$ for all $t \geq t_0$. Consistent with fault-assumptions in the consensus literature, we will assume that it is the adversary who solely decides which members of G crash and when.

3.1.1 Assumptions about Node Connectivity

Consider two operative members that are in the wireless range of each other. Let δ be the maximum delay that any one of them may take to transmit an application message to the other, despite possible collisions and interferences. The members are said to be *1-hop connected* or simply *1-Connected* at time $t \geq t_0$, if they remain operative and also in each other's range at least until $(t+\delta)$. Note that *1-Connectivity* at time $t \geq t_0$ is a binary relation on $W(t + \delta)$ which is reflexive, symmetric and intransitive.

Two operative members, N_i and N_j, are said to be *h-Connected* if a path of at most h, $h \geq 1$, *1-Connections*, exists between them for the next $h\delta$ duration at least. N_i and N_j are said to be *(B,h)-Connected* if they are operative and *h-Connected* for a duration of length at least B, where $B > h\delta$ is a parameter specified by the application.

The intuition behind *(B,h)-Connectivity* is that *h-connectivity* between two members is useful to an application, only if it lasts for an additional duration of at least $(B - h\delta)$ time. Applications that are of interest here are the multicast protocols used by the consensus protocol. Note that, by definition, *(B,h)-Connectivity* will imply *(B,h′)-Connectivity* for $h' > h$ if $B + (h' - h)\delta \approx B$, i.e., if δ is negligibly small compared to B. If *(B,h)-Connectivity* $\Rightarrow$ *(B,h′)-Connectivity*, then *(B,h)-Connectivity* is a reflexive, symmetric and transitive relation.

3.1.2 Liveness Requirement

From an application's perspective, a member N_i is permanently isolated from the rest of G starting from time t, if it never enjoys *(B,h)-Connectivity* with any operative member for any h, $h \geq 1$, at any time after t. Of course, the above notion of member isolation assumes that N_i remains operative and that the absence of *(B,h)-Connectivity* emergence is not because N_i crashed sometime after t.

Solving consensus requires that operative members are not isolated permanently; the MANET must fulfill a *liveness condition* that eliminates such permanent member isolations. This condition can be informally stated as follows. For every t, $t \geq t_0$, any $N_i \in W(t)$ that remains operative for a 'long enough' duration will have *(B,h)-Connectivity* for some h, $h \geq 1$, with some operative member N_j before $t+I_h$, where I_h, $I_h \geq B$, is finite but unknown.

Observe that there are two unknowns in the informal statement of the liveness condition: h and I_h. To keep the protocol design tractable, we assume that only I_h is unknown and the condition holds for *all* h, $h \geq 1$. More precisely, our assumption will be as stated below:

For every t, $t \geq t_0$, any $N_i \in W(t)$ that remains operative for a 'sufficiently long' time will make *(B,h)-Connectivity*, for every h, $h \geq 1$, with some operative member before $t + I_h$, where I_h, $I_h \geq B$ is finite but unknown.

When *(B,h)-Connectivity* is assured for all $h \geq 1$, applications can choose to work with a particular value of h, say H. Note that if H is chosen to be small, *(B,H)-Connectivity* may take longer to emerge. That is, the smaller the H, the more likely that I_H is large and the more delay-tolerant the application needs to be.

Formally, the liveness condition assumed can be stated as:

$$\begin{aligned} &\forall t \geq t_0, \forall h \geq 1, \exists I_h, B \leq I_h \neq \infty : \\ &\quad (\forall N_i \in W(t+I_h), \exists N_j, N_j \in W(t+I_h) : \\ &\qquad N_i \text{ and } N_j \textit{ (B,h)-Connect} \text{ at some time in } [t, t+I_h]) \end{aligned} \tag{3.1}$$

In words, for every t, $t \geq t_0$, for every $h \geq 1$, there exists a finite but unknown I_h, $I_h \geq B$, such that for every $N_i \in W(t + I_h)$ there exists $N_j \in W(t + I_h)$ which *(B,h)-Connects* with N_i at some time during the interval $[t, t + I_h]$.

Node isolation is a special case of group partitioning in which a subset, say, G', of operative members are unable to *(B,h)-Connect* with any of the operative member not in G'. To avoid G' from being permanently partitioned, the MANET must allow some member in G' to *(B,h)-Connect* with some operative member in (G-G') before $t + I_h$, for all $h \geq 1$. So generalizing (3.1) gives the required Liveness Condition that is assumed to be satisfied by the MANET so that G is never permanently partitioned from the application's perspective:

Liveness Condition (LC)

$$\begin{aligned}
&\forall t \geq t_0, \forall h \geq 1, \exists I_h, B \leq I_h \neq \infty : \\
&\quad \forall G', W(t+I_h) \supset G' \neq \{\}, \exists N_i, N_j : \\
&\qquad (N_i \in G', N_j \in W(t+I_h) - G' \ \wedge \\
&\qquad N_i \text{ and } N_j \textit{ (B,h)-Connect} \text{ at some time in } [t, t+I_h])
\end{aligned} \tag{3.2}$$

In words, for every t, $t \geq t_0$, for every $h \geq 1$, there exists a finite but unknown I_h, $I_h \geq B$, such that for every non-empty $G' \subset W(t+I_h)$ the following holds: there are operative members N_i and N_j such that $N_i \in G'$, $N_j \in W(t+I_h) - G'$ and *(B,h)-Connectivity* exists between them starting from some time in the interval [t, $t+I_h$].

Remark. $I_h = \infty$, $\forall h \geq 1$ and $I_H = B$, for some finite H, represent two extreme cases of interest.

When $I_h = \infty$, $\forall h \geq 1$, an application will never see correct members having *(B,h)-Connectivity* for durations that are as long as it needs. That is, it is pointless for the application to be delay indulgent because the MANET is never going to oblige its requirement.

On the other hand, $I_H = B$ means that, at every t, new *(B,H)-Connectivity* between some $N_i \in G'$ and some $N_j \in W(t+I_h) - G'$ is emerging, or an existing *(B,H)-Connectivity* prolongs beyond t for a further B time or more, or both. That is, the MANET keeps the operative members of G always *(B,H)-Connected*.

3.1.3 Categories of MANETs Depending on Node Connectivity

Chapter 1 shows three types of MANETs; dense, DTN and sparse. The node connectivity is low in DTN and sparse MANETs, but there is a distinction between these two types; In DTN, two operative members N_i and N_j might never make *(B,h)-Connectivity*, for any h ($h \geq 1$), with each other even if they stay operative. So sending a message between two members requires any intermediate node to store the message until this node joins connectivity with either the destination member or another intermediate node. On the other hand, members in sparse MANETs satisfy the liveness condition (3.2) which contains three variables B, h and I_h:

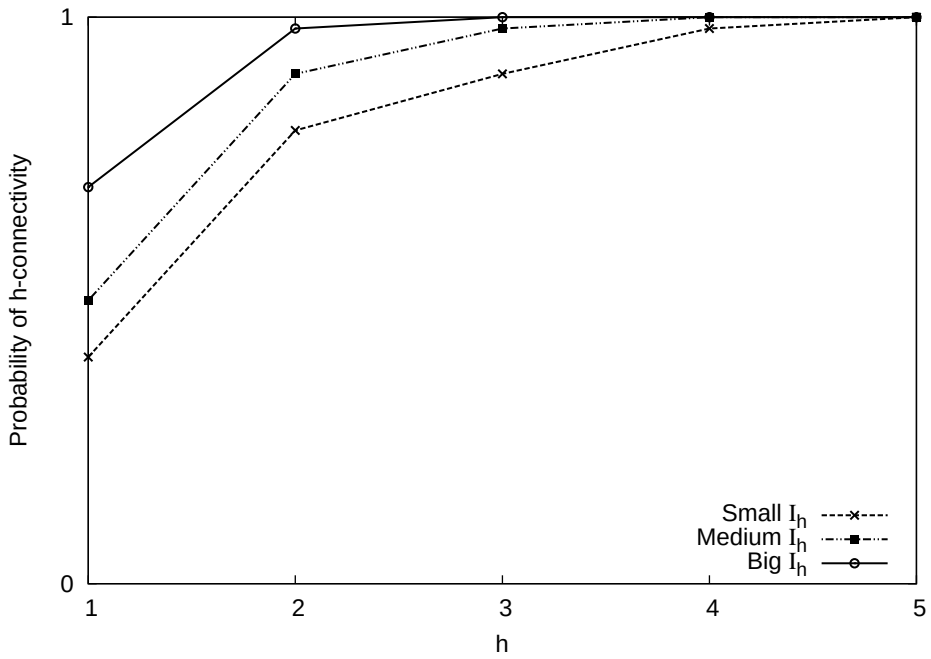

Figure 3.1: Probability of h-connectivity in dense MANETs

1. The value of B is mainly decided by the application. Applications of interest here are the multicast protocols. So we will explain about the value of B when we introduce our multicast protocols.

2. The value of h can be any finite number greater than 0, so applications can select a particular value of h. There have been several studies which try to choose an optimal value for h. In [KDPH05] and [jLBrP03] the protocols try a set of values before selecting a suitable value for h. The detailed study of choosing the optimal value of h is kept beyond the scope of this thesis. The only requirement for the system is to satisfy the liveness condition for any $h \geq 1$. Note that choosing a small value for h, say H, might cause the *(B,H)-Connectivity* to take longer time to emerge. So the smaller the H, the more delay tolerant the application needs to be.

3. The value of I_h represents the interval of time during which the *(B,h)-Connectivity* between any two members will start.

Figures 3.1, 3.2 and 3.3 show a qualitative comparison between dense, sparse and DTN MANETs using different values of h and I_h. This comparison shows that:

- In dense MANETs: The probability of the *(B,h)-Connectivity* between any two members is high even when the value of I_h is not big; Increasing the value of

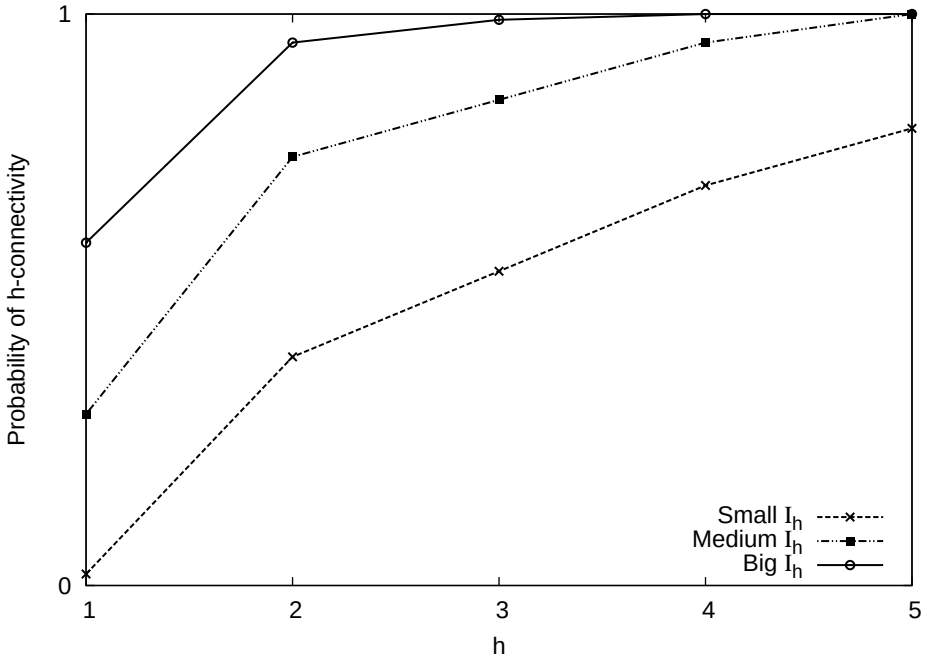

Figure 3.2: Probability of h-connectivity in sparse MANETs

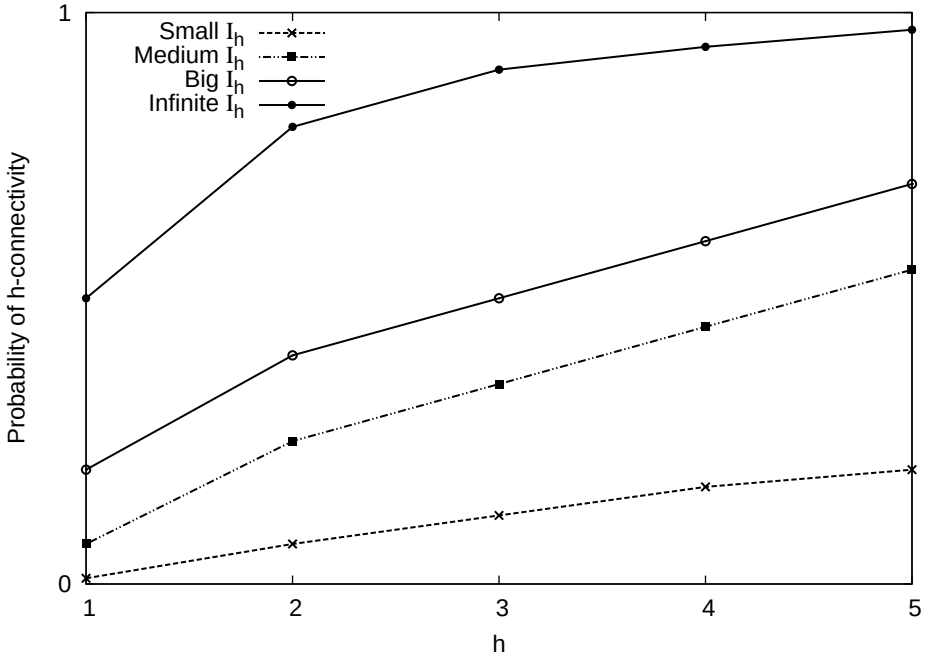

Figure 3.3: Probability of h-connectivity in DTN MANETs

I_h gives higher probability of the *(B,h)-Connectivity*. Moreover, incrementing the value of h always results in higher probabilities of *(B,h)-Connectivity* (see Figure 3.1).

- In sparse MANETs: Figure 3.2 shows that a high probability of *(B,h)-Connectivity* requires big values of I_h even for moderate values of h (e.g. h=3). So, using a small value of I_h results in low probability of *(B,h)-Connectivity* despite

depending on big values of h (e.g. h=5).

- In DTN MANETs: The probability of *(B,h)-Connectivity* is low even for big values of I_h and h. Moreover, when the value of I_h grows without bound ($I_h \rightarrow \infty$) and for big values of h, the probability cannot be guaranteed to reach 1 because there may exist pairs of members which might never join *(B,h)-Connectivity* (see Figure 3.3).

Figure 3.4 shows how to obtain a probability of *(B,h)-Connectivity* equals to 1 in sparse MANETs: It indicates that a big value of h is required when I_h is small and bigger value of I_h can guarantee the connectivity even when smaller values of h are used.

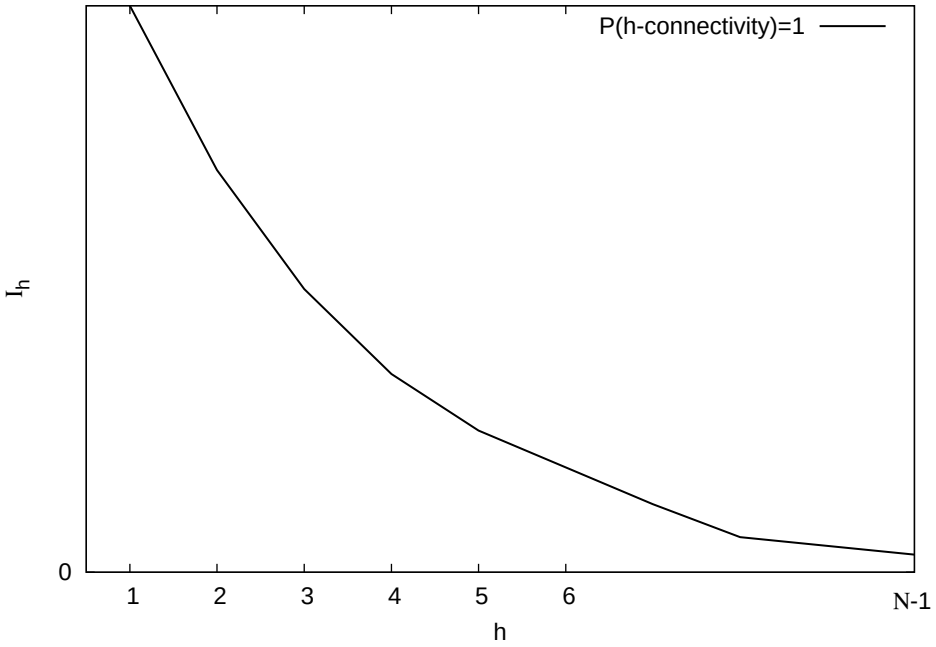

Figure 3.4: values of I_h for P=1 in sparse MANETs

3.1.4 Multicast Services

The authors of [BEV06] describe the eventually relinquishing and eventually quiescent properties of broadcasting protocols. We will define these properties for multicasting protocols.

A multicast protocol is said to be *eventually relinquishing* ($\Diamond R$), if it ensures that there is a time after which operative members do not retain a multicast message for dissemination purposes.

A multicast protocol is said to be *eventually quiescent* ($\diamondsuit Q$, for short), if it ensures that there is a time after which operative members permanently stop disseminating a multicast message.

So in a multicast protocol with $\diamondsuit R$ property, each member eventually stops retaining a message m and also, by implication, stops sending m. On the other hand, a multicast protocol with $\diamondsuit Q$ property guarantees only eventual cessation of sending of m; especially, it might require the members to retain m for ever. The latter discussion proves that (i) $\diamondsuit R \Rightarrow \diamondsuit Q$, and (ii) $\diamondsuit Q$ does not necessarily imply $\diamondsuit R$ because $\diamondsuit Q$ might retain m for ever.

Basic flooding and its variations, such as [HOTV99], are both $\diamondsuit Q$ and $\diamondsuit R$. However, they cannot *guarantee* anything on the number or the set of members that would receive a multicast message. We will use the term *coverage* to refer to the number or the type of members that receive a multicast message. We will define two classes of protocols ($\diamondsuit RC$ and $\diamondsuit QC$) that offer the maximum possible guarantee on coverage while also being $\diamondsuit R$ or $\diamondsuit Q$ respectively.

In expressing the protocols' properties (and also throughout the thesis), we will use the term *receive* to refer to a member taking possession of a message by executing a protocol. We will use m to denote a message; we will also assume that a member that initiates multicasting of m also receives its own m. The term *transmit* will be used for referring to a member's MAC level, wireless transmitting of a packet that can be *received* by all devices (members and non-members) that are in the wireless range.

Definition.

$\diamondsuit RC$ is a class of multicast protocols which satisfy:

1. Members that receive m discard m within some finite time after multicast of m is initiated ($\diamondsuit R$).

2. In any execution in which at least one correct member receives m, at least $(n - f)$ members receive m (*coverage*, C).

Remark 1: All members that receive m need not be correct, and at most f can go on to crash after receiving m. So, in the worst case, just $(n-2f)$ correct members may receive m. It is shown below that the 2 above is the strongest coverage guarantee any $\diamondsuit R$ protocol can offer.

Lemma 1 *In any execution in which at least one correct member receives m, a $\diamondsuit R$ multicast protocol that has no support for detecting member crashes, cannot guarantee that more than $(n-f)$ members receive m if more than $(n-f)$ members are operative throughout that execution.*

Proof By Contradiction. Suppose that there exists a $\diamondsuit R$ protocol that cannot detect member crashes but nevertheless guarantees that more than $(n-f)$ members receive m if more than $(n-f)$ members are operative throughout. Let $\mathcal{F}$ be any set of f members and $\overline{\mathcal{F}} = G - \mathcal{F}$. We consider two executions of this protocol and some correct member in $\overline{\mathcal{F}}$ multicasts m at time t_b in both the executions.

Execution 1: All members of $\mathcal{F}$ have crashed before t_b. Let t_e, $t_e > t_b$, be the timing instance before which any member that receives m has relinquished m. t_e must exist since the protocol is $\diamondsuit R$; also, no more than $(n-f)$ members can receive m.

Execution 2: No member crashes before or after t_b. However, the MANET keeps all members of $\mathcal{F}$ outside the wireless range of every member in $\overline{\mathcal{F}}$ at least until t_e. This is possible if, in this execution, the unknown $I_h > t_e - t_b$ for every $h \geq 1$; that is, LC is met for $I_h > t_e - t_b$.

Nodes of $\overline{\mathcal{F}}$ cannot distinguish execution 2 from execution 1 until t_e for three reasons: (1) the multicast initiator is in $\overline{\mathcal{F}}$, (2) members of $\mathcal{F}$ do not execute the protocol for m (certainly until t_e in execution 2), and (3) members of $\overline{\mathcal{F}}$ cannot detect whether a member in $\mathcal{F}$ is crashed or operative.

So, in execution 2, as in execution 1, only at most $(n-f)$ members can receive m with all receiving members discarding m by t_e; additionally, members of $\mathcal{F}$ do not execute the protocol until t_e and there is no m after t_e for them to receive. This is a contradiction, since at least f members do not receive m even though all n members are operative.

$\square_{Lemma\ 1}$

Definition.

$\diamondsuit QC$ is a class of multicast protocols which satisfy:

1. Dissemination of m stops permanently within some finite time after multicast of m is initiated ($\diamondsuit Q$).

2. In any execution in which at least one correct member receives m, all correct

members receive m (*coverage*, C).

Remark 2: Executions in which no correct member receives m involve members crashing shortly after receiving m and not executing the protocol long enough for m to reach a correct member. They have $\diamondsuit Q$ and $\diamondsuit R$ features, by default. So, we will limit our interest to those executions in which at least one correct member receives m.

Remark 3: No multicast protocol can offer a stronger coverage guarantee than 2 above, unless it also implements *uniform* delivery property. The latter requires delaying delivery of m until it is deduced that at least $(f+1)$ members can receive m. This incurs time and bandwidth overhead.

As per [ACT00], transforming a $\diamondsuit QC$ protocol into a $\diamondsuit R$ equivalent with the same coverage guarantee as 2, is not possible unless operative members can accurately know which members have crashed. Such a knowledge is in turn not possible because I_h in equation (3.2) can be arbitrarily long. So, a crashed member is indistinguishable from an operative one that is out of other members' wireless range for an arbitrarily long period of time. So, 2 must be weakened for the $\diamondsuit R$ counterpart.

When comparing between $\diamondsuit RC$ and $\diamondsuit QC$, we have the following observations:

1. When a correct member receives m: $\diamondsuit QC$ can guarantee that all correct members will receive m, but $\diamondsuit RC$ can guarantee that at least $(n-2f)$ correct members will receive m. So, in terms of coverage, $\diamondsuit QC \Rightarrow \diamondsuit RC$.

2. $\diamondsuit RC \Rightarrow \diamondsuit QC$ in terms of freeing buffers because $\diamondsuit RC$ deletes a message m, and also stops sending m, within some finite time after the multicast of m is initiated and $\diamondsuit QC$, however, only stops sending m.

3.1.5 Approach to Consensus and the Rationale

Given that every operative member in G proposes an initial value or *initial estimate*, a consensus protocol satisfies the following properties which lead to members arriving at an identical decision despite their potentially different initial estimates:

- **Validity**: If a member decides v, then v was proposed by some member.
- **Agreement**: No two members decide differently.
- **Termination**: Every correct member decides.

As was discussed in Chapter 1, consensus is typically solved using multicast protocols that assume that the network remains connected always (or almost always), and assure that correct members receive each other's messages with a probability that is 1 (or close to 1, respectively). Using these consensus solutions as such for sparse MANETs would involve deploying a $\diamondsuit QC$ protocol for multicasting. This could lead to an unaffordable storage overhead for two reasons: the $\diamondsuit QC$ protocols can retain messages for ever and consensus may take several rounds of message exchange, using a large number of $\diamondsuit QC$ multicasts.

An alternative approach, which is pursued here, will be to use a $\diamondsuit RC$ multicast protocol and address the challenges that arise thereof. In particular, a $\diamondsuit RC$ protocol, in the worst case, can deliver a correct member's m just to $(n - 2f)$ correct members (Remark 1 in Subsection 3.1.4); consequently, if $(n-2f)$ is not a majority in n, which will be the case if $n \leq 4f$, then executions of a traditional consensus protocol may deadlock. Choosing the alternative approach requires addressing these issues.

It is also not entirely feasible to rule out the use of a $\diamondsuit QC$ protocol. When a correct member decides during an execution of a consensus protocol, it is common for another correct member not to be able to decide at or around the same time; the latter needs to be 'helped' by the former by sending the decision to it. A $\diamondsuit QC$ protocol, will be used (only) for this purpose. Figure 3.5 depicts the role of $\diamondsuit QC$ and $\diamondsuit RC$ protocols in our approach.

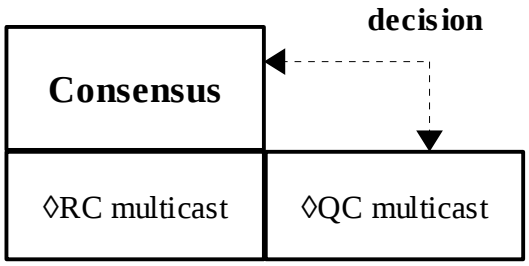

Figure 3.5: Multicast Support

3.2 Summary

This chapter introduced the system model for MANETs. Three main points were defined accurately in this chapter:

1. The requirements on the network connectivity which are essentail for solving the consensus: These requirements state that operative members do not get isolated permanently. So the liveness condition, that eliminates such permanent member isolations, was introduced.

2. The principles of required multicast protocols for solving the consensus: Two types of multicast protocols, $\diamondsuit QC$ and $\diamondsuit RC$, were defined.

3. Our approach for solving the consensus using $\diamondsuit QC$ and $\diamondsuit RC$ protocols.

Chapter 4

Encounter Gossip Multicast Protocol

The flooding protocol is one of the most common protocols for MANETs. This protocol works in a very simple manner: (i) When a node N_i initializes a message m it transmits m for one time (ii) If a node N_j receives m for the first time, it transmits this message after a small random wait (redundant copies of a received message will be ignored). So it is clear that the flooding protocol does not require any information about the network topology or the routes between nodes. This makes this protocol to be convenient for MANETs where the network topology changes, due to node mobility, are very frequent. However, It was proved in Chapter 1 that the flooding protocol performs poorly in low densities. The authors of [HOTV99] explains more about the problems of the flooding protocol for MANETs.

The Encounter Gossip *EG* broadcast protocol [CEM09] is a persistent form of flooding. This protocol performs well in a wide range of node speeds and network densities including the low densities. Moreover, *EG* preserves the same flexibility of the flooding protocol with the regard of the network topology-independence. This protocol is $\diamondsuit R$ because nodes delete any message m after transmitting m for a limited number of times.

We use the *EG* protocol to derive our Encounter Gossip Multicast *EGM* protocol. So this chapter will first present the *EG* protocol, then the *EGM* protocol will be introduced.

4.1 Encounter Gossip EG Broadcast Protocol

As was mentioned, this protocol is a persistent form of flooding. The main focus of this protocol is to improve the coverage (the fraction number of nodes which receive a message m) of the flooding protocol. This protocol depends on the notion of encounter and transmits a message m a number of times upon encounters. Note that the protocol presented in this part is a broadcast one, so all nodes show the same interest in receiving each message m ($G = \mathcal{S}$).

4.1.1 Protocol Definition

Describing this protocol requires defining the following terms:

- Beacon: Every node N_i needs to send its ID periodically, this ID can be carried in a small packet (called beacon). So the beacon tells other nodes 'hello, I am node N_i', by sending this beacon every node in the system can advertise its existence to other nodes in the area.

- Neighborhood list: When a node N_i receives the beacon from another node N_j, it realizes that N_j is within its wireless range, so N_i adds N_j to its neighborhood list $Neigh_i$. Thus this list contains all neighbors identifiers $Neigh_i = \{N_j, N_k, N_l, ...\}$ whose beacons have been recently received by N_i. Node N_i prunes $Neigh_i$ by removing any node from whom few executive beacons (3 in our implementation) are missing.

- Encounter: Node N_i encounters another node N_j when N_j, which is not in the neighborhood list of N_i, is added to $Neigh_i$. In this case, it is not necessary for N_j to encounter N_i at the same time because the beacon packets are not synchronized between nodes. Note that, if N_j leaves the neighborhood list of N_i and later enters it again this will be considered as a fresh encounter.

- Threshold τ: A message must be transmitted a total of τ times to ensure high expected coverage. This requirement can be met by having every node transmits a message τ times after encounters. τ is estimated to be

$$\tau = 2\lceil \ln N + \gamma \rceil \tag{4.1}$$

Where N is the total number of nodes in the system, $\gamma = 0.5772...$ is the Euler-Mascheroni's number.

According to this broadcast protocol, each node in the system acts as follows:

1. When a node N_i initiates or receives a new message m it creates a counter $c(m)$ which is set to 0, then N_i stores m associated with $c(m)$. If m is received from another node N_j, the node N_i adds N_j to its neighborhood list.

2. Upon having a new initiated (or received) message m, if node N_i has any neighbors (apart from the sender of m), it transmits m once and increases $c(m)$ by 1.

3. When node N_i encounters any other node, it checks $c(m)$, if $c(m) < \tau$ it transmits m once and increments $c(m)$ by 1.

4. Node N_i keeps transmitting m upon encounters until $c(m) = \tau$, after that it removes m from the memory.

5. When node N_i removes a message m from the memory it keeps the ID of m. So in the future, if m is received again it will be ignored.

4.2 Encounter Gossip Multicast EGM Protocol

The *EG* broadcast protocol is used to send messages to all nodes in the system. So *EG* treats G as $\mathcal{S}$ itself. On the other hand, in case of multicast only a subset of nodes $G \subset \mathcal{S}$ need to exchange messages. So the *EG* broadcast protocol needs to be adapted properly to enable only nodes in G to exchange messages upon encounters. The next section explains about the steps which are required for changing the *EG* protocol to become a multicast protocol.

4.2.1 Approach

As was explained earlier, in case of broadcast where all nodes are interested in receiving messages, *EG* protocol performs well. According to *EG*, all nodes in the system transmit messages upon encountering each others. On the other hand, when there is only a subset of nodes G exchange messages, every member $N_i \in G$ can have two types of encounters:

1. Group encounter: It happens when a member $N_j \in G$ is added to $Neigh_i$.

2. Non-group encounter: It occurs when a node $X_i \in (\mathcal{S} - G)$ is added to $Neigh_i$.

A simple adaptation of EG protocol for $G \subset \mathcal{S}$, will be to allow a node $N_i \in G$ to transmit m only when it encounters another $N_j \in G$ in its $Neigh_i$, and do nothing if the encounter involves a node $X_i \in (\mathcal{S} - G)$. But it could turn out to be highly delay-indulgent when $|G| \ll |S|$. That is, a node $N_i \in G$ needs to wait for a long time before encountering any $N_j \in G$. So, we allow occasional flooding of m. N_i initiates a flooding of m if the number of consecutive encounters that did not involve any $N_j \in G$ exceeds a threshold called the encounters-to-flood and denoted as ϕ. The value of ϕ has to be chosen carefully because a big value of ϕ might cause a long delay and a small value might cause excessive flooding. We will assume that a flood initiated by a member reaches at least one other member, just like in an encounter-triggered transmission.

4.2.2 Protocol Definition

This protocol transmits messages by using two different ways; flooding and encounter gossip transmission (see Figure 4.1). A node N_i upon receiving a message m should be able to recognize in which way m was transmitted (flooding or encounter gossip transmission). Therefore, the flooding messages will be marked differently by the source nodes. The following section explains about the two ways used to transmit messages and when each of them is applied.

4.2.2.1 Flooding

This part deals with the received messages which were sent by flooding and it works according to the following steps: When any node receives a message m, which has not been forwarded by this node, (i) it forwards a copy of this message, and (ii) if the receiving node is a group member, m is passed to the encounter gossip part of this node (see figure 4.1); otherwise, m is discarded.

4.2.2.2 Encounter Gossip Based

Encounter gossip based requires group members to have some data structures.

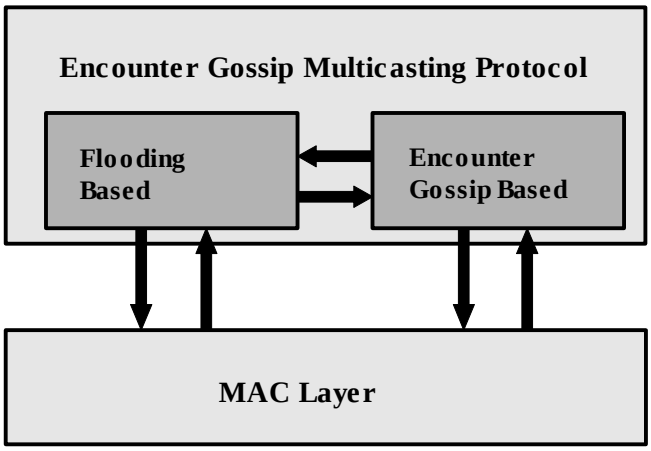

Figure 4.1: Encounter Gossip Multicast *EGM* Protocol disseminates messages using flooding and encounter based

4.2.2.2.1 Data Structures The protocol defines three additional data structures, in addition to retaining $c(m)$ which was described earlier

1. *c(m)*: Counts the number of encounter transmissions and floods which a member has carried out for multicasting m.

2. *encounters(m)*: Keeps track of the number of consecutive encounters involving only non-members. When it reaches ϕ, a flood of m is initiated and *encounters(m)* is reset to 0. So its value at any moment during multicast of m will be in [0, ϕ].

3. Propagation History for m, $PH(m)$: A member node maintains the most recent values of $c(m)$ which the member nodes (including itself) are known to have had. It is made up of 2-tuples of the form $<N_i, c>$ one for a distinct member node N_i whose $c(m)$ is most recently known to have had the value c.

4. The integer variable, group propagation counter for m, $gc(m)$, indicates the total number of transmissions or floods that are known to have occurred for multicasting of m; $gc(m) = \sum_{M} c$ for all $<M, c>$ in $PH(m)$.

5. Encounter History for m, $EH(m)$: A member N_i maintains, in EH, the ID of any member N_j which has flooded or transmitted m to N_i.

These data structures are maintained only by member nodes. Note also that maintaining $PH(m)$ requires piggybacking a copy of it onto every m that is being

transmitted or flooded; and, updating the $PH(m)$ using the piggybacked information in a received m. Since $c(m)$ of a group member cannot decrease, updating of $PH(m)$ should choose the biggest c for a given member N_i; if $<N_i, c>$ is present in the piggyback. Finally, a member node halts the execution when its $gc(m) \geq \tau$. Therefore, a piggyback $PH(m)$ will not have more than τ tuples in it and τ is *O(n ln(n))*.

4.2.2.2.2 Protocol Behavior It consists of two parts and it is described for a member N_i handling multicast message m.

Part 1:

- When N_i initializes multicasting of m, it initializes $c(cm)$, $gc(m)$, and *encounters(m)* to zero; $PH(m)$ and $EH(m)$ to empty. If $Neigh_i$ contains one or more members, then N_i enters $<N_i, c(m) + 1>$ into $PH(m)$, piggybacks $PH(m)$ onto m and transmits m.

- If N_i receives m for the first time it initializes $c(m)$, $gc(m)$, and *encounters(m)* to zero; $PH(m)$ to the piggybacked $PH(m)$ of the received m and $EH(m)$ to the ID of the source member of m.

Part 2: The following two concurrent tasks are carried out until $gc(m) \geq \tau$

- On receiving m: $PH(m)$ is updated with the piggybacked $PH(m)$ of the received m and the ID of the source member of m is added to $EH(m)$.

- On encountering a node: (i) If the encountered node is not a member *encounters(m)* is incremented by 1; If *encounters(m)*$\geq \phi$, it is set to 0, m is prepared, a flood of prepared m is initiated, and $c(m)$ is incremented by 1. (ii) If the encountered node is a member: (a) if the encountered member is not in EH, m is prepared and the prepared m is transmitted, and (b) *encounters(m)* is set to 0 and $c(m)$ is incremented by 1.

Preparing of m for flooding/transmission involves replacing the tuple $<N_i, c>$ in $PH(m)$ with $<N_i, c(m) + 1>$ and piggybacking $PH(m)$ onto m.

4.2.2.3 Protocol Interface

The *EGM* protocol offers four primitives which can be invoked for multicasting a message m. These primitives are shown in Figure 4.2 and explained below:

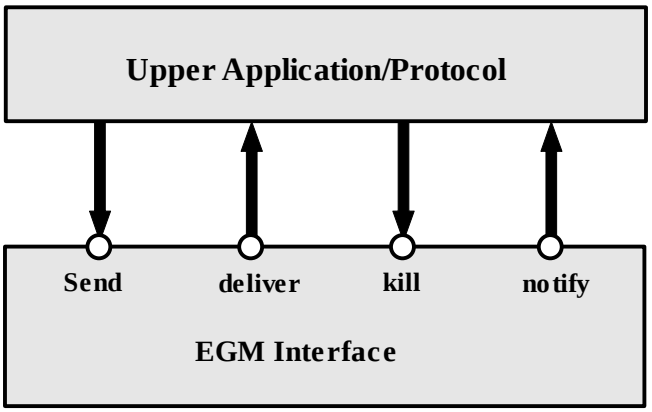

Figure 4.2: The *EGM* interface includes the functions provided to the upper protocol

1. *send*: This function is invoked when the upper application (or the member) has any message m to send to other members. So the upper level orders *EGM* protocol to send m by invoking *send(m)*.

2. *deliver*: When *EGM* receives a message m it can pass m to the upper application by invoking *deliver(m)*.

3. *kill*: It is invoked by the upper level to order *EGM* to stop dealing with a message m. This happens when *EGM* is working to send m then the upper level decides to stop handling m. In fact, this function adds more flexibility to this protocol because it allows the upper level to interfere at any time to stop sending any message.

4. *notify*: When *EGM* finishes sending a message m, it informs the upper application by invoking *notify(m)*.

4.2.3 Performance Study

This section studies the performance of the *EGM* protocol through simulation. All simulations in this thesis were performed using JiST/SWANS simulator, a Java based discrete event simulator [BHvR05]. The simulation parameters and the metrics used to evaluate the protocol performance will be, shortly, explained in details.

4.2.3.1 Simulation Parameters

The default simulation parameters used throughout this thesis, unless otherwise specified, are shown in Table 4.1. 50 mobile nodes of $\mathcal{S}$ were randomly placed in a fixed size terrain of 1000m x 1000m. The multicast group G contains 10 nodes which are chosen randomly at the start of each experiment and they do not leave the group during the experiment. The network density was varied by varying the nodes' wireless range as 100, 150 and 200 meters, resulting in the density values 1.6, 3.5 and 6.3 respectively. The mobility model for nodes is Random Waypoint: In this model, each node chooses a destination randomly and starts moving to this destination; once the node reaches its destination, it pauses for a chosen interval, it again randomly picks out a new destination and so on.

Table 4.1: Default simulation parameters

Simulation Parameters	
Simulator	JiST/SWANS[BHvR05]
Area size	1000m x 1000m
Mobility style	Random Waypoint
Pause time	0s
Node placement	Random
$\|\mathcal{S}\|$	50
$n = \|G\|$	10
Wireless range	100m, 150m, 200m
Density	1.6, 3.5, 6.3
Maximum Node Speed	5 m/s - 40 m/s
Pathloss model	Free-Space

4.2.3.2 Performance Metrics and Observations

It is important to define the following factors and metrics:

- Total multicasts: Is the total number of messages to be multicasted by group members.
- Average of coverage: Is the average of destinations that receive a given multicast:

$$\text{average of coverage} = \frac{1}{\text{total}} \left(\sum_{i=1}^{total} c_i \right) \tag{4.2}$$

Where total refers to the total multicasts, and

$c_i = \frac{\text{destinations that received the multicast } m_i}{\text{total number of destinations for } m_i}$

- Average of group cost: Is the average number of messages transmitted by each member for a given multicast:

$$\text{Avg group cost} = \frac{\text{number of transmissions by all members}}{\text{total multicasts} * |G|} \tag{4.3}$$

- Average of floods: Is the average number of initialized floods by a group member for a given multicast:

$$\text{Avg of floods} = \frac{\text{total number of all initialized floods}}{\text{total multicasts} * |G|} \tag{4.4}$$

- Average of the total cost: Is the average number of transmissions committed by any node for a given multicast:

$$\text{Avg of total cost} = \frac{\text{total number of transmissions by all nodes}}{\text{total multicasts} * |\mathcal{S}|} \tag{4.5}$$

- Average of group response time: Group response time can be defined as the interval between generating a message and the moment when all members have stopped propagating it. So the average of group response time is:

$$\text{average of group response time} = \frac{1}{\text{total}} \left(\sum_{i=1}^{total} res_i \right) \tag{4.6}$$

  Where res_i = group response time for the multicast m_i.

A simulation for a particular set of parameters involves 100 runs using distinct random seeds. Thus, a point in the graphs we present is the average on measurements taken over 100 runs. Moreover, each run commenced after 1000 seconds of node movement to avoid any initial bias in node placement.

In this part, each member sends 10 messages of 512 Bytes each and, to avoid losses due to collisions, sent messages (from any member) were well spaced out in time.

We study the effect of using different values of τ which had the values 1, 2, 4, 6, 8, and 10. For each value of τ we used different values of ϕ as 0, 2, 4, 6, and 8 to show the effect of flooding on the protocol behaviour.

Figures 4.3 and 4.4 show the group coverage achieved as a function of the encounters threshold τ using maximum node speeds 5 m/s and 10 m/s with node densities

of 1.6 and 6.3 respectively. They show that increasing τ causes the group coverage to improve; (i) at low densities, increasing τ causes substantial improvement of group coverage, and (ii) at high densities, the effect of increasing τ is correspondingly smaller. The second observation from these Figures is about the effect of ϕ:

1. At low densities: When τ is less than 4, smaller values of ϕ cause better group coverage. This means that the flooding is more effective than the encounter based transmission when τ is less than 4. On the other hand, the encounter based transmission becomes more effective than the flooding when $\tau \geq 4$.

2. At high densities: Smaller values of ϕ provide better group coverage until $\tau = 6$ where the effect of ϕ disappears for $\tau \geq 6$.

Comparing Figures 4.3 and 4.4 shows that the group coverage is always higher when densities are higher (as expected).

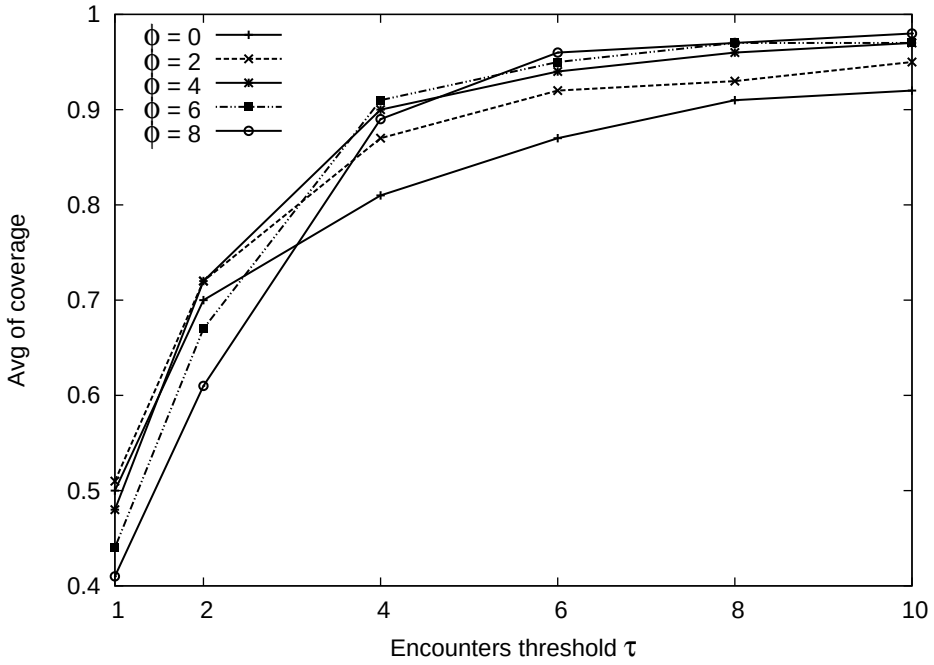

Figure 4.3: The group coverage vs the encounters threshold τ, Wireless range = 100m, node density = 1.6, max speed = 5m/s

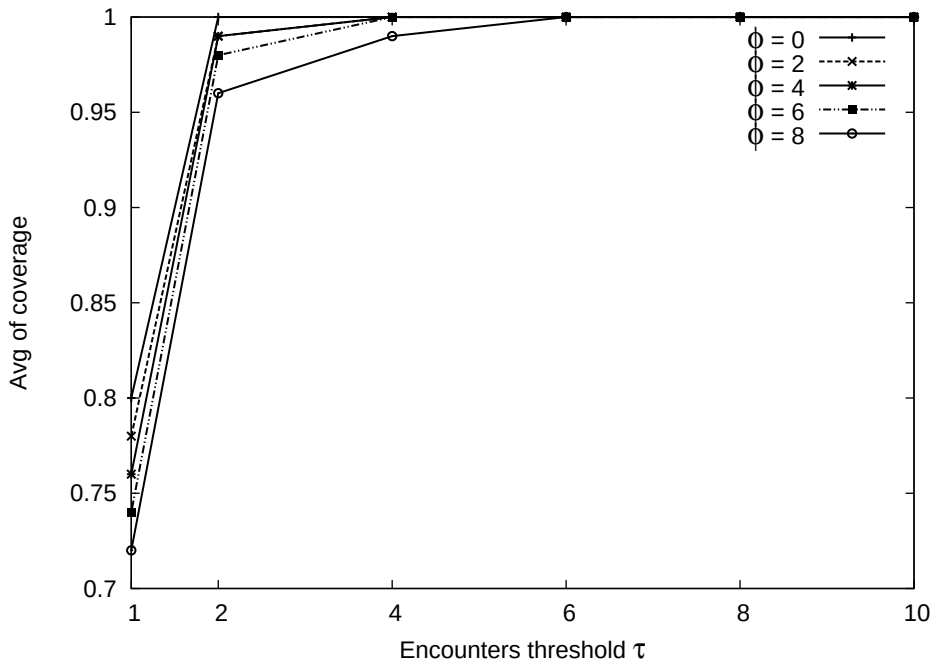

Figure 4.4: The group coverage vs the encounters threshold τ, Wireless range = 200m, node density = 6.3, max speed = 10m/s

There are several factors which might affect the number of initiated floods by group members. These factors include the value of τ, the value of ϕ, group ratio (the ratio of group members $|G|$ to the total number of nodes $|\mathcal{S}|$), node density and node speed. In this part we will concentrate on the effect of the values of τ and ϕ. So figures 4.5 and 4.6 show the effects of the values of τ and ϕ on the average of initiated floods, by a group member, for a given multicast. In case of low densities, increasing τ always results in increasing the number of floods (as expected). As when densities are high, the number of floods increases dramatically up to a point ($\tau = 2$), and then a slight increase is experienced. This is because when ($\tau > 2$), the $PH(m)$ is diffused quickly due to the high group coverage (see Figure 4.4) so $gc(m)$ grows quickly to reach τ and therefore members halt sending m. Another observation is that the number of floods in all densities is increased by decreasing the value of ϕ (as expected).

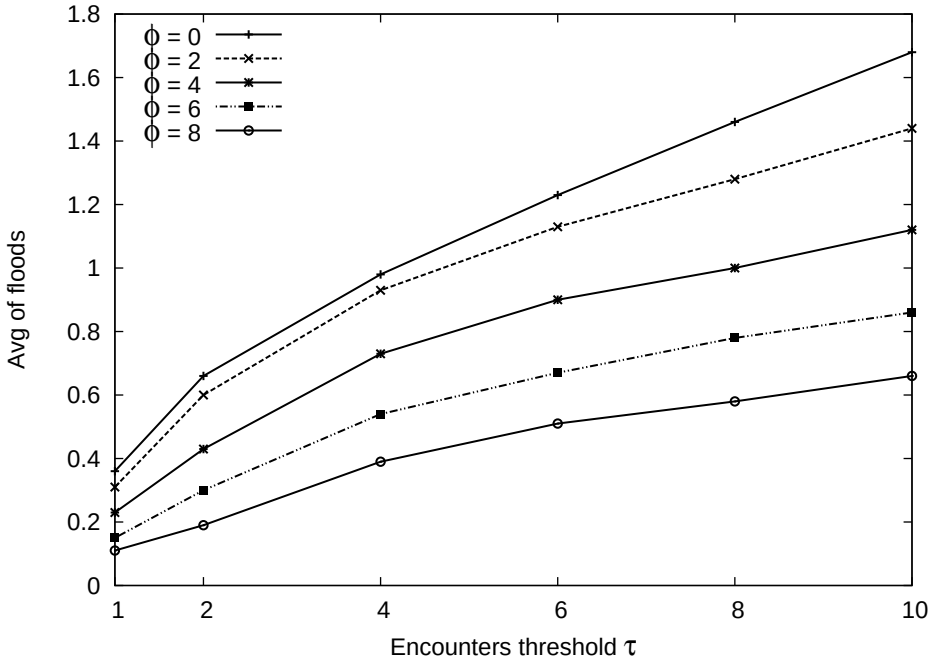

Figure 4.5: The average of initiated floods by a group member for a given multicast vs the encounters threshold τ. Wireless range = 100m, max speed = 5m/s

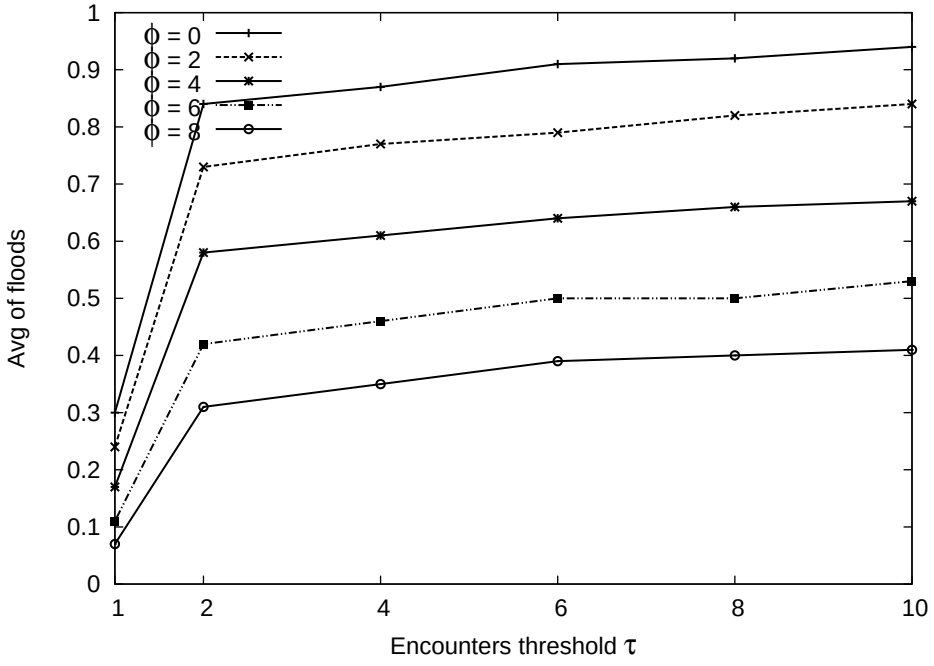

Figure 4.6: The average of initiated floods by a group member for a given multicast vs the encounters threshold τ. Wireless range = 200m, max speed = 10m/s

Figures 4.7 and 4.8 show the transmission cost by a member for a given multicast. Figures 4.9 and 4.10 show the total transmission cost by a node (member and non-member) for a given multicast. These graphs show the same trend as the flooding graphs (4.5 and 4.6) for low and high densities. Note that at high densities, increasing τ after the point ($\tau = 2$) has a small effect on the transmission cost.

This is because, as was explained, the group coverage is very high after this point so $PH(m)$ and $EH(m)$ can be diffused quickly to prevent any redundant transmissions.

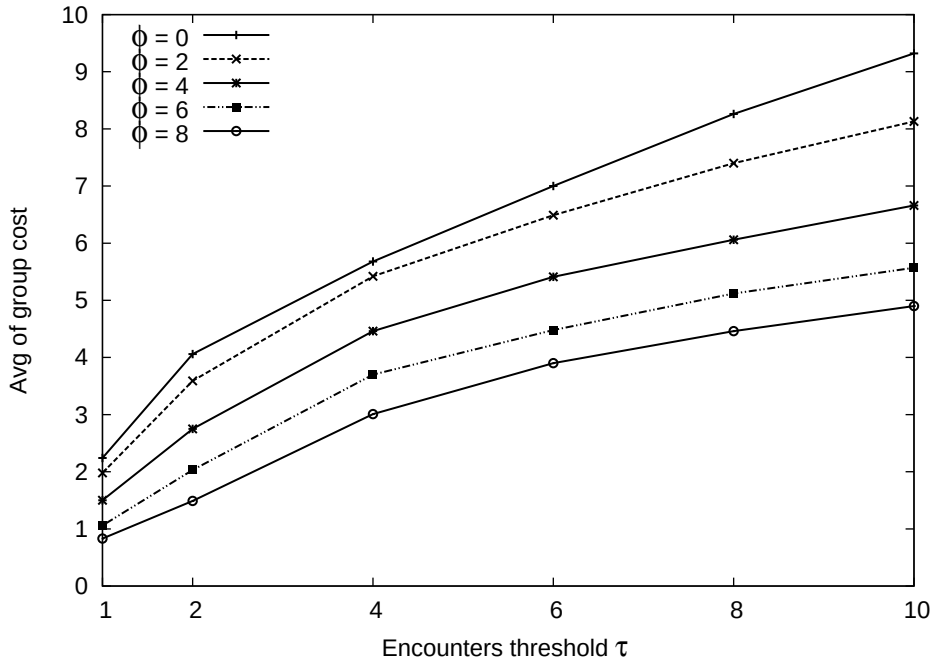

Figure 4.7: The average of transmitted messages per member for a given multicast vs the encounters threshold τ. Wireless range = 100m, max speed = 5m/s

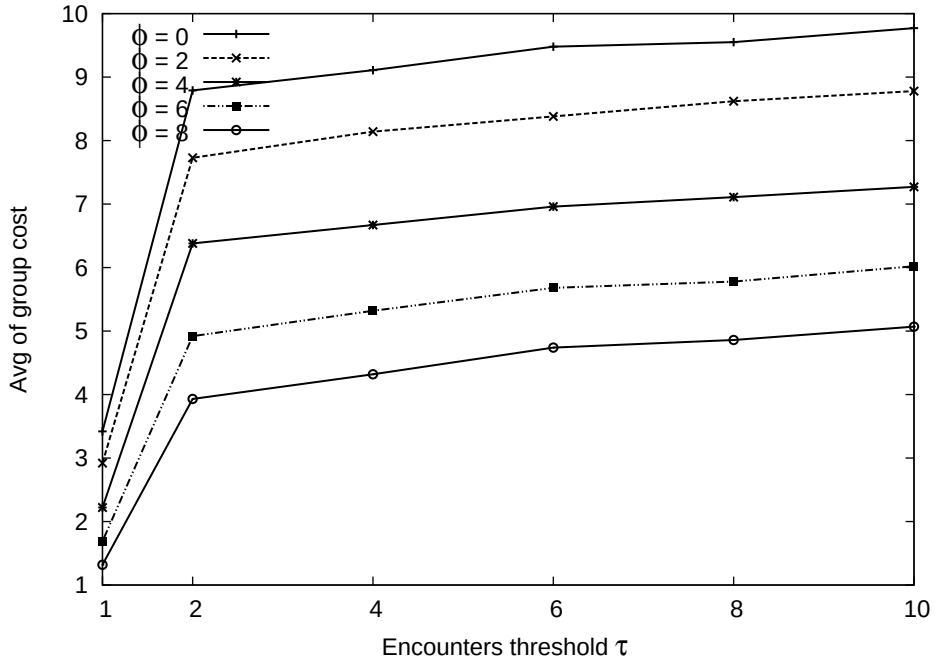

Figure 4.8: The average of transmitted messages per member for a given multicast vs the encounters threshold τ. Wireless range = 200m, max speed = 10m/s

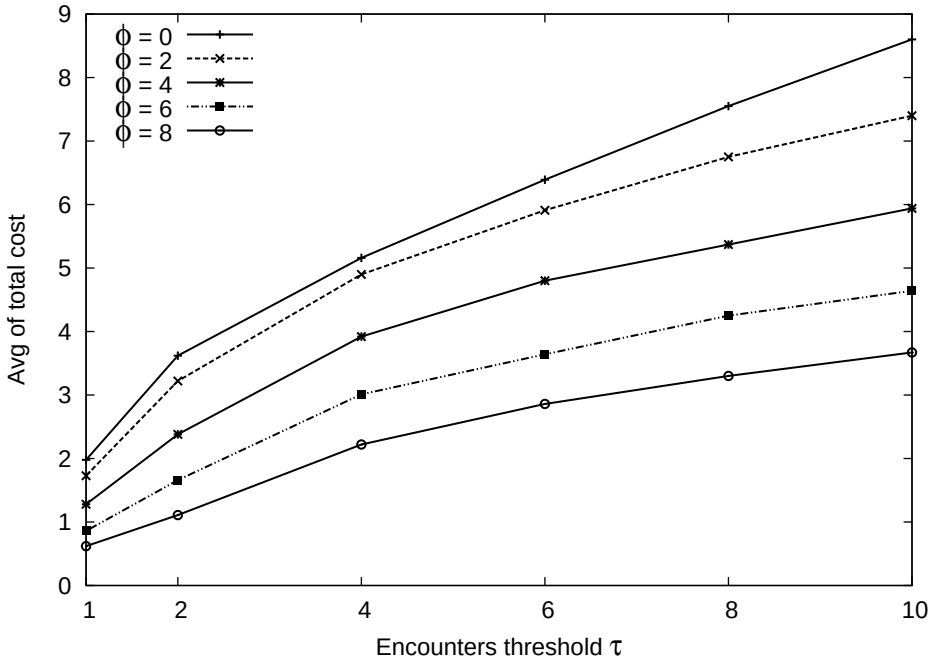

Figure 4.9: The average number of transmissions per a node vs the encounters threshold τ. Wireless range = 100m, max speed = 5m/s

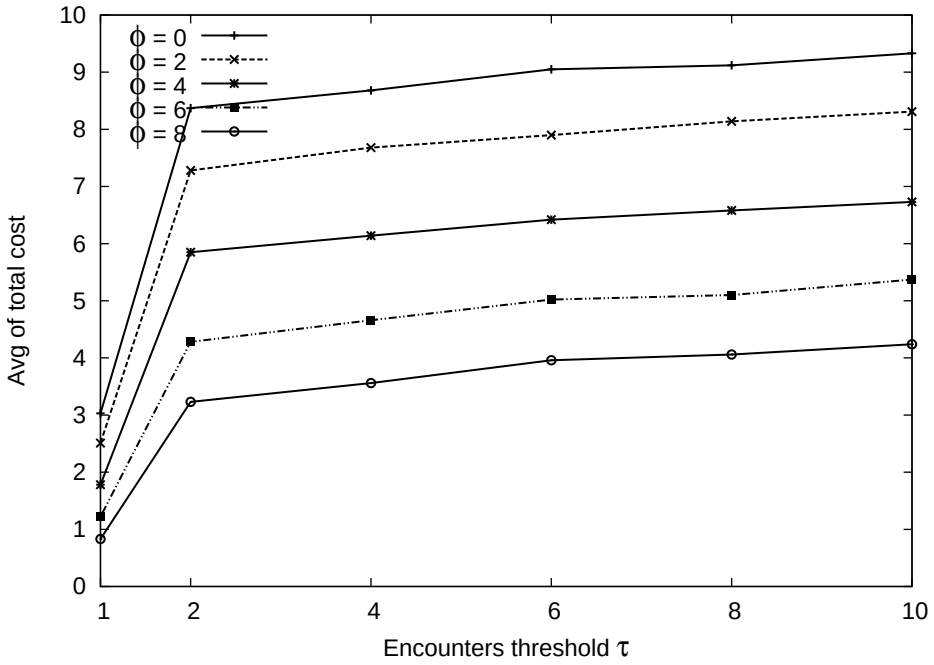

Figure 4.10: The average number of transmissions per a node vs the encounters threshold τ. Wireless range = 200m, max speed = 10m/s

Figures 4.11 and 4.12 show the average of the group response time. These graphs show that using more floods (smaller values of ϕ) reduces the response time considerably. This is because when members use more floods, they can escape waiting members encounters to occur. Moreover, increasing the node density helps reduce

the group response time. This is because increasing the node density results in higher connectivity which makes members encounter each other more quickly.

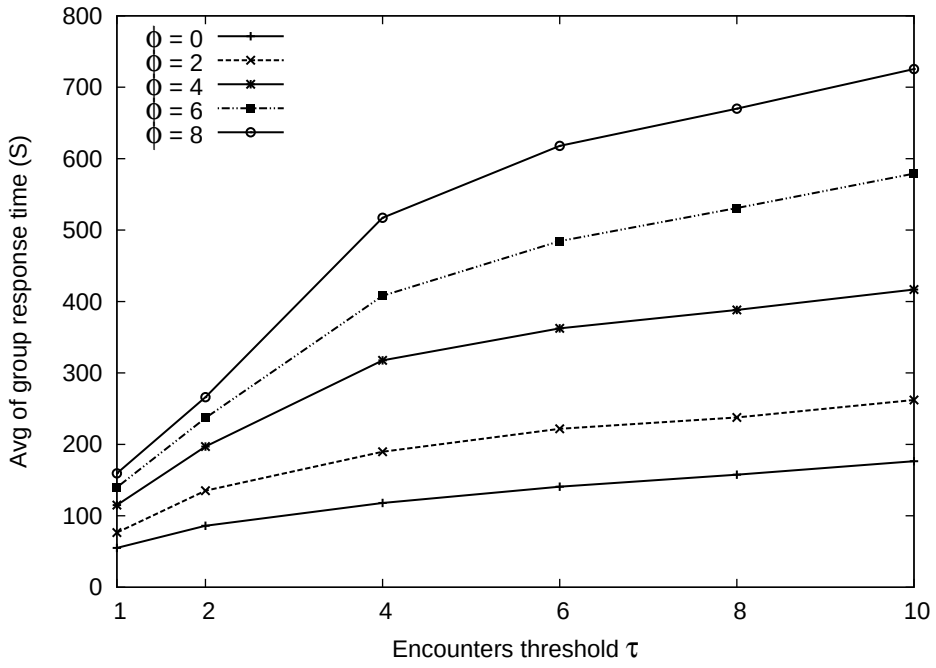

Figure 4.11: The group response time (second) vs the encounters threshold τ. Wireless range = 100m, max speed = 5m/s

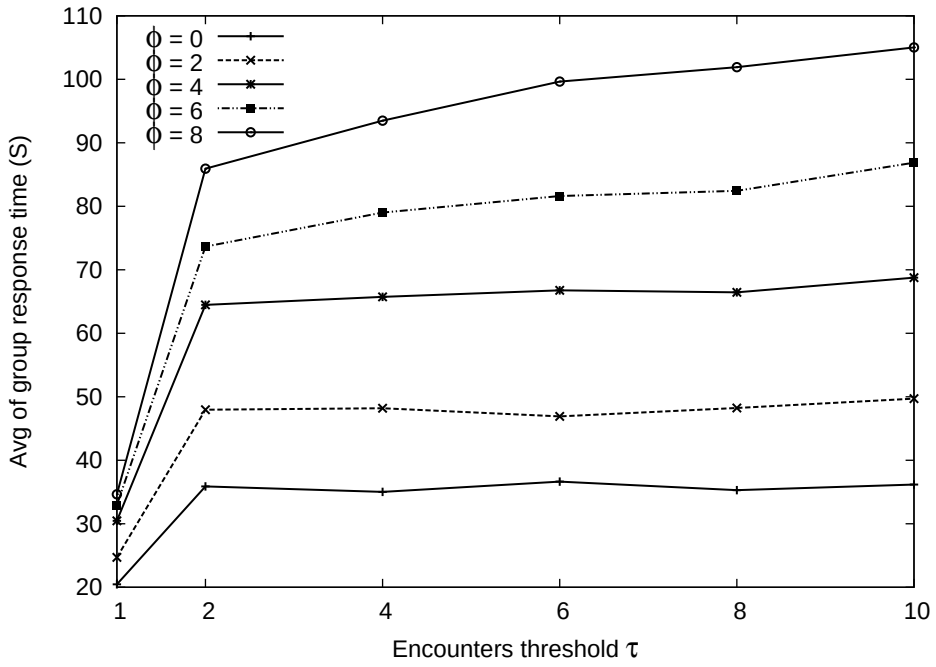

Figure 4.12: The group response time (second) vs the encounters threshold τ. Wireless range = 200m, max speed = 10m/s

4.3 Optimization of EGM Protocol

The flooding was added to this protocol mainly to reduce the delay when $|G| \ll |S|$. The last section has showed the effect of the value of ϕ on coverage, cost and delay. It was shown that a small value of ϕ increases the transmission cost (see Figures 4.9 and 4.10). On the other hand, using a big value of ϕ increases the protocol response time (or the delay) (see Figures 4.11 and 4.12). Therefore, we will introduce an optimization for *EGM* to prevent members from repeating any unnecessary floods. This optimization, which is called *flood reduction* optimization, adapts the value of ϕ according to some factors which can decide the need for flooding. This optimization considers two factors; node speed and node density. So the next section will explain about the effect of these two factors on the need of flooding.

4.3.1 Effects of Node Speed

There are two obersvations about the effect of node speed:

1. When the node speed is low: The traveled distance by nodes will be small. So encounters between members are less likely to occur. Therefore, members need to use flooding to be able to deliver their messages as the encounter based transmission will be rare.

2. If nodes move more quickly (high speed): Members are more likely to encounter each others so they can exchange messages upon encounters. Therefore, using the flooding should be kept to minimum.

4.3.2 Effects of Node Density

The node density has a substantial effect on the efficiency of flooding:

1. At high node density: When a member N_i floods a message m, this message is more likely to be received by other members because the network connectivity is high. So any further flooding of m by N_i is more likely to be redundant becuase it is assumed that members have already received m. Therefore, N_i should avoid flooding the same message m in the future.

2. At low node density: When a member N_i floods a message m, no/few members are expected to receive m because the connectivity between nodes is low. So N_i

should avoid repeating the flooding of m because this flooding is not effective.

4.3.3 *flood reduction* Optimization

In the original *EGM* protocol, the value of ϕ does not change during the protocol execution, but according to this optimization ϕ will be changed depending on some events. So, the *flood reduction* will adaptively vary the value of ϕ, similar in style to additive increase and multiplicative decrease in TCP congestion control, so that flooding is rarely resorted. Therefore, this optimization adds the following two steps to the *EGM*:

- ϕ is initialized when the protocol starts: When nodes move slowly, ϕ is initialized to a small value and as the node speed increases, ϕ is increased accordingly.
- After a member N_i transmits a message m:
  1. If N_i is transmitting m by flooding, it doubles the threshold ϕ (ϕ=2ϕ).
  2. If N_i is transmitting m by encounter based, ϕ takes the value $\phi = \max \{\phi - 1, \textit{encounters(m)}+1\}$.

4.3.4 Performance Study

This section studies the performance of the optimized *EGM* where the value of ϕ is adapted according to the *flood reduction* optimization. We use the same simulation parameters in Table 4.1. Moreover, all simulation parameters and factors used in studying *EGM* performance stay the same, that is, except the values of ϕ and the range of maximum node speeds. So, each member sends 10 messages and each simulation involves 100 runs using distinct random seeds. The value of ϕ is initialized depending on the node maximum speed and it is increased when this speed is increased. So all simulations which have the same node maximum speed start by having the same value of ϕ, and during execution the value of ϕ is adapted when any transmission occurs (see *flood reduction* optimization steps). Table 4.2 shows the initial values of ϕ.

We also show the effect of using different values of τ. However, when evaluating the coverage and cost we will concentrate on the point $\tau = 6$. This value of τ is estimated to give a coverage levels close to 1. That is, according to Equation 4.1 after replacing N by $n = |G| = 10$ (in case of multicast).

Table 4.2: Initial values of ϕ

Maximum Speed (m/s)	**Initial Value of ϕ**
5	2
10	3
15	4
20	5
25	6
30	7
35	8
40	9

Figures 4.13 - 4.15 show the group coverage achieved using different values of encounters threshold τ. The first observation is that the node speed has, mostly, no effect on the group coverage. That is, despite the use of different values of ϕ for different maximum speeds. The second observation is about the group coverage for $\tau = 6$, which reaches at least 0.95 in all, low and high, densities.

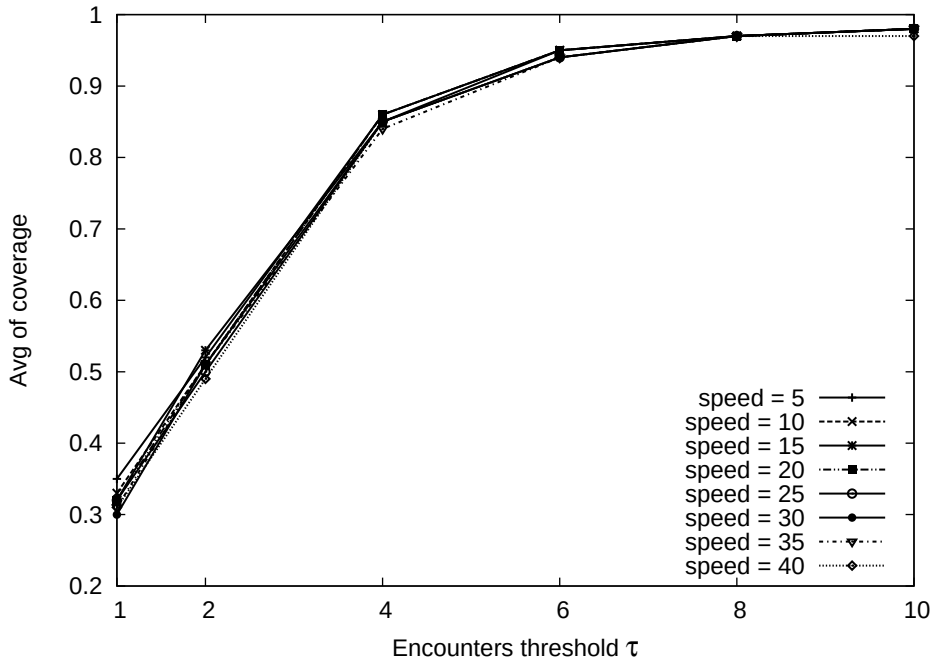

Figure 4.13: The group coverage vs the encounters threshold τ, Wireless range = 100m, node density= 1.6

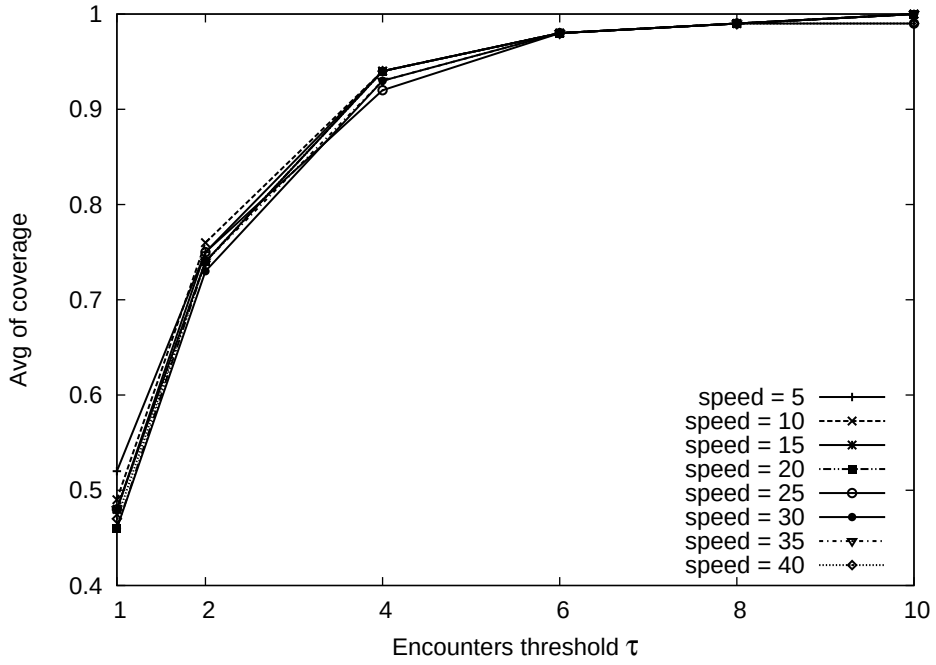

Figure 4.14: The group coverage vs the encounters threshold τ, Wireless range = 150m, node density= 3.5

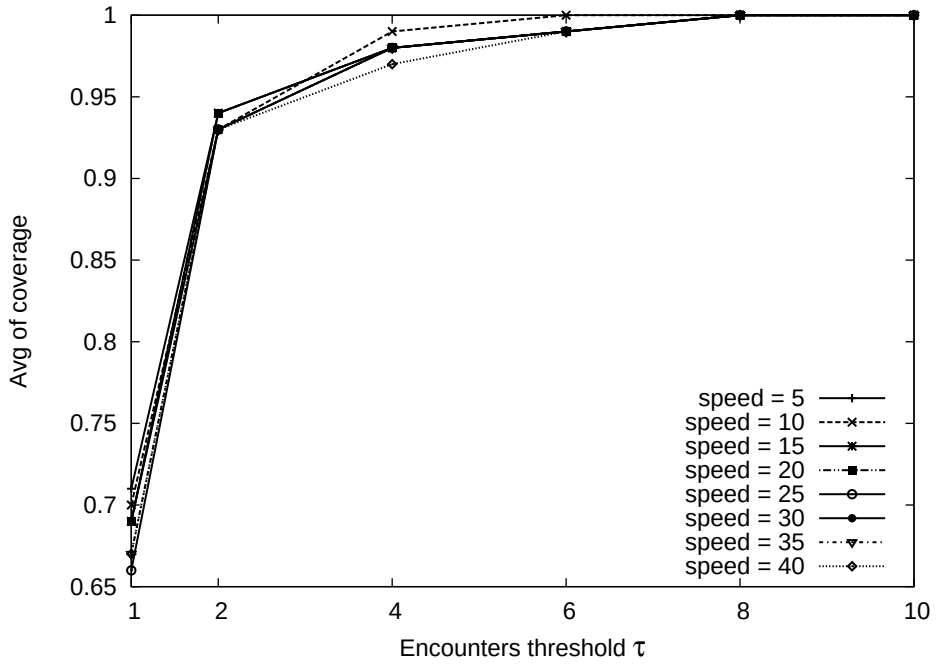

Figure 4.15: The group coverage vs the encounters threshold τ, Wireless range = 200m, node density= 6.3

Figures 4.16 - 4.18 show the effect of different values of τ and maximum node speed on the average of initiated floods. These Figures show that the maximum node speed has no considerable effect on the initiated floods. Moreover, it is clear that the average number of initiated floods is substantially reduced here comparing with the original *EGM*. This can be observed by comparing Figures 4.16 and 4.18

with the correspondent Figures in the original *EGM* (see Figures 4.5 and 4.6). Note that during this comparison, we choose an average value for ϕ, say $\phi = 4$, from the latter Figures.

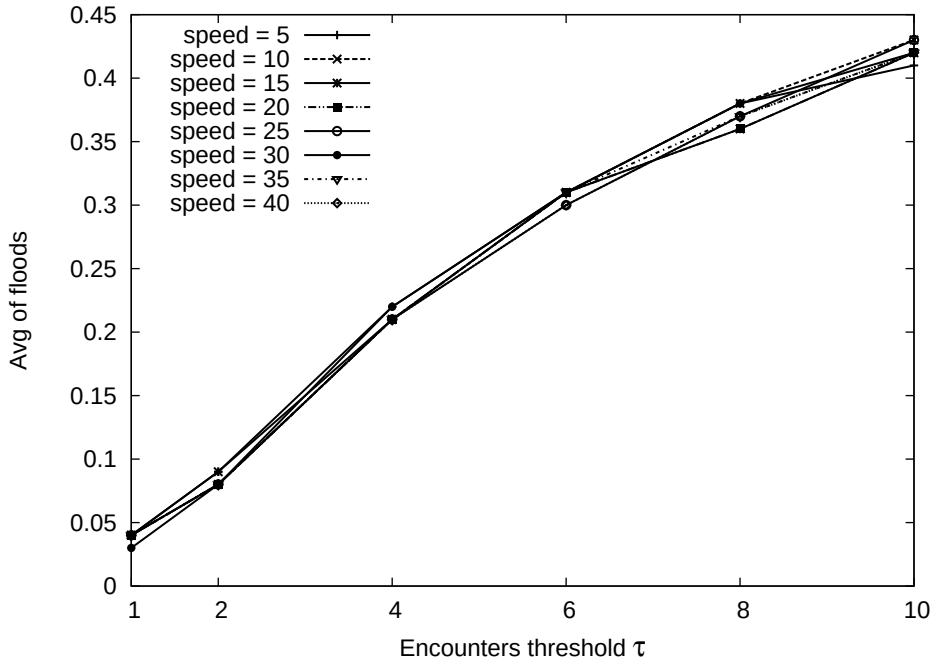

Figure 4.16: The average of initiated floods by a group member for a given multicast vs the encounters threshold τ. Wireless range = 100m

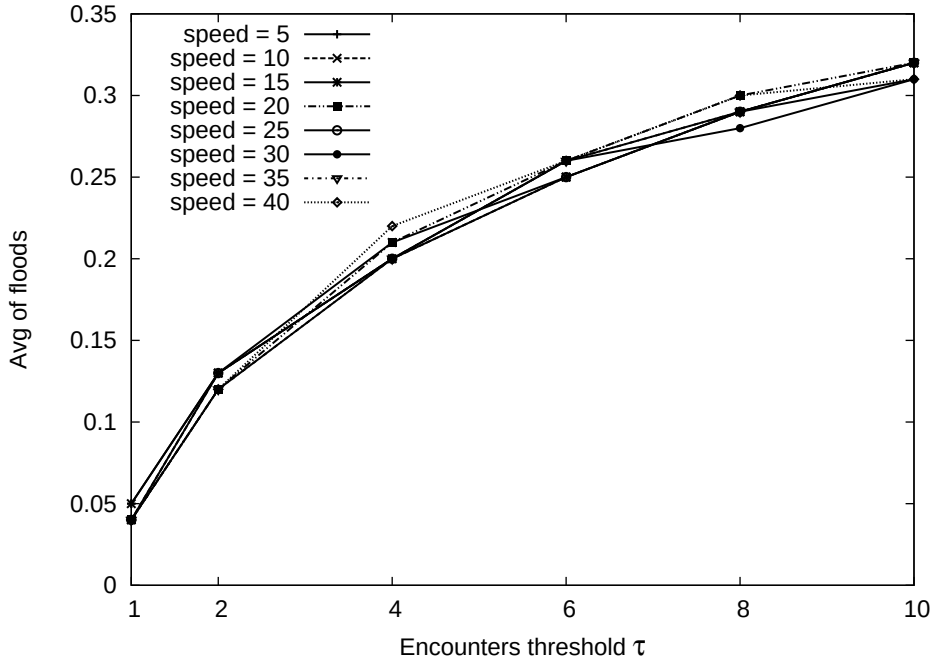

Figure 4.17: The average of initiated floods by a group member for a given multicast vs the encounters threshold τ. Wireless range = 150m

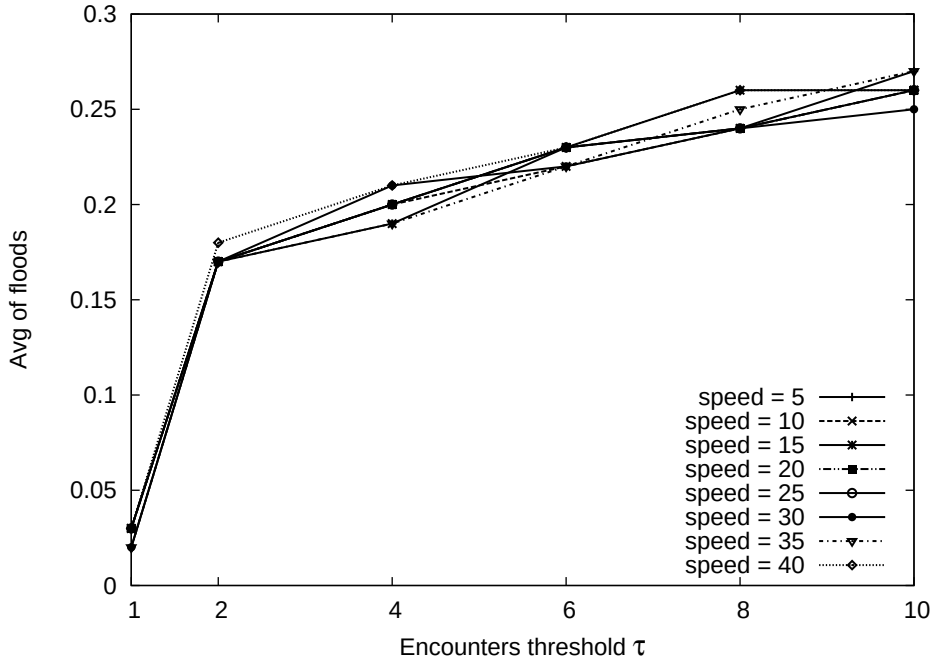

Figure 4.18: The average of initiated floods by a group member for a given multicast vs the encounters threshold τ. Wireless range = 200m

Figures 4.19 - 4.21 show the transmission cost by a member and Figures 4.22 - 4.24 show the transmission cost by a node (member and non-member). These Figures again show that the maximum node speed has no effect on the transmission cost. Moreover, we compare Figures 4.19 and 4.21 with the transmission cost by a member in the original *EGM* and Figures 4.22 and 4.24 with the transmission cost by a node in the original *EGM*. This comparison shows that the transmission cost in the optimized *EGM* is reduced considerably. Note also that we choose $\phi = 4$ from the correspondent Figures in the original *EGM*.

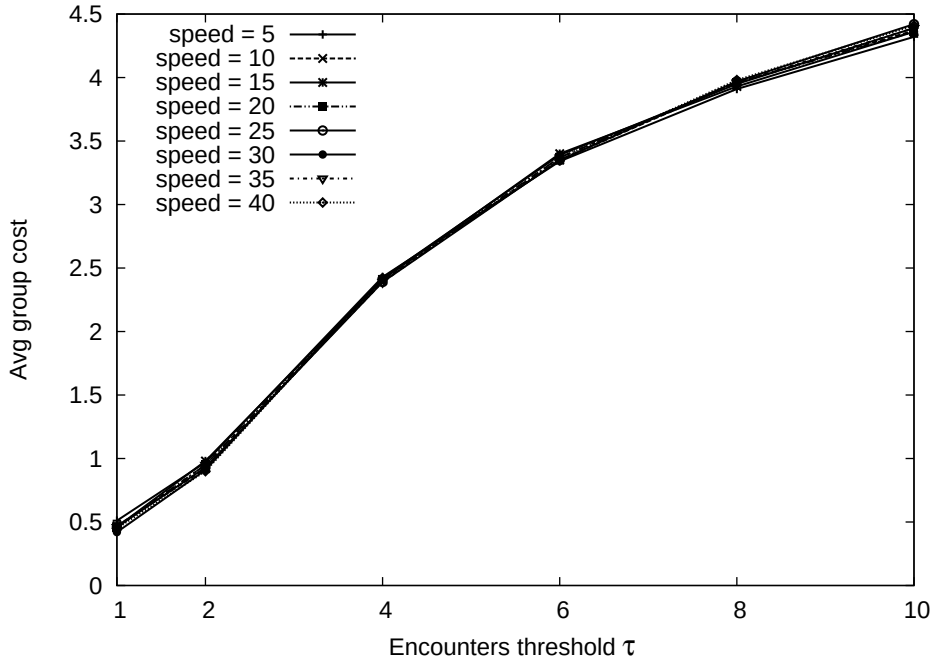

Figure 4.19: The average of transmitted messages per member for a given multicast vs the encounters threshold τ. Wireless range = 100m

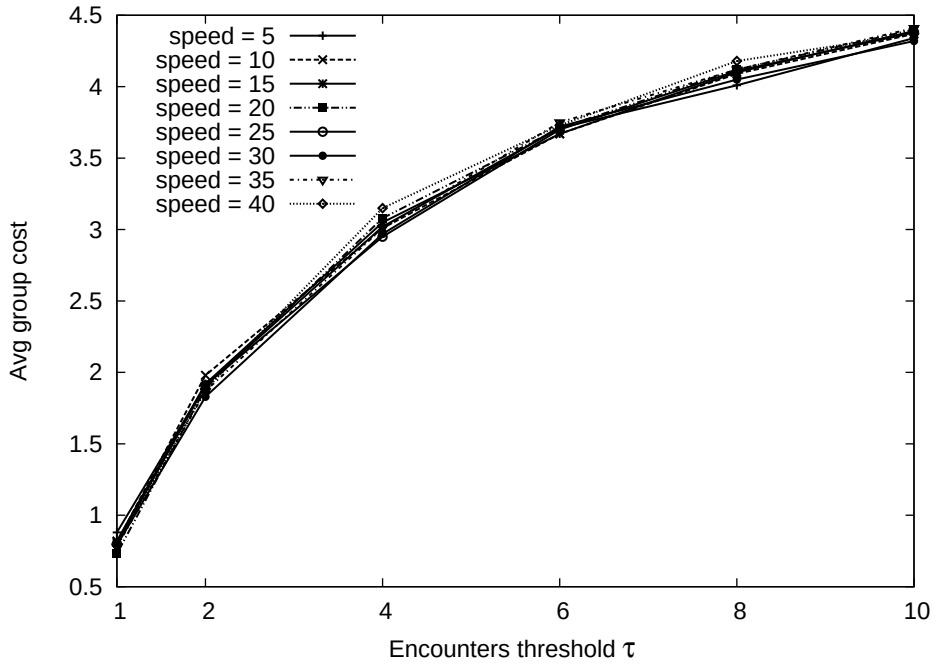

Figure 4.20: The average of transmitted messages per member for a given multicast vs the encounters threshold τ. Wireless range = 150m

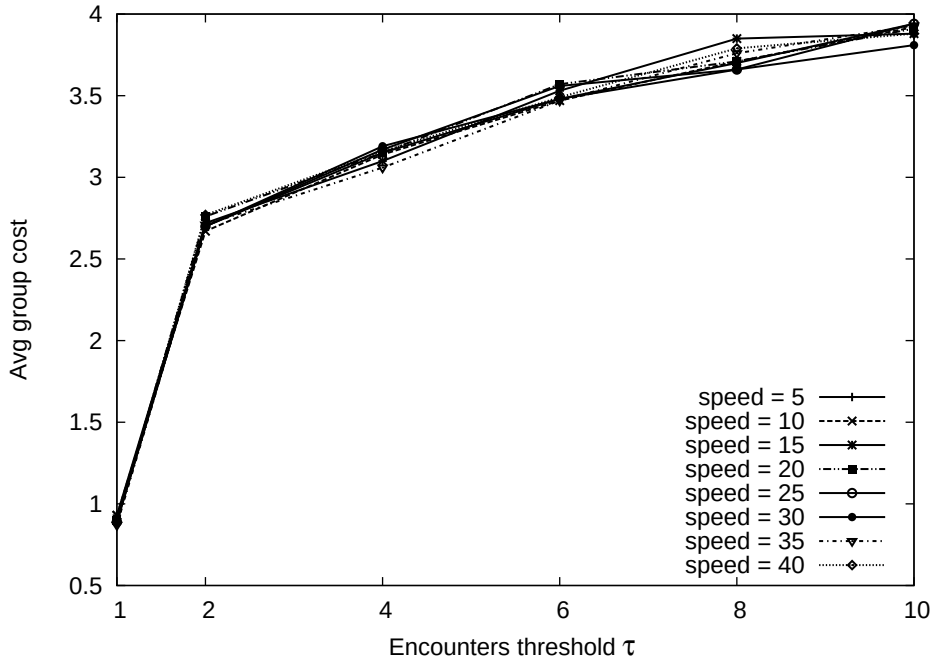

Figure 4.21: The average of transmitted messages per member for a given multicast vs the encounters threshold τ. Wireless range = 200m

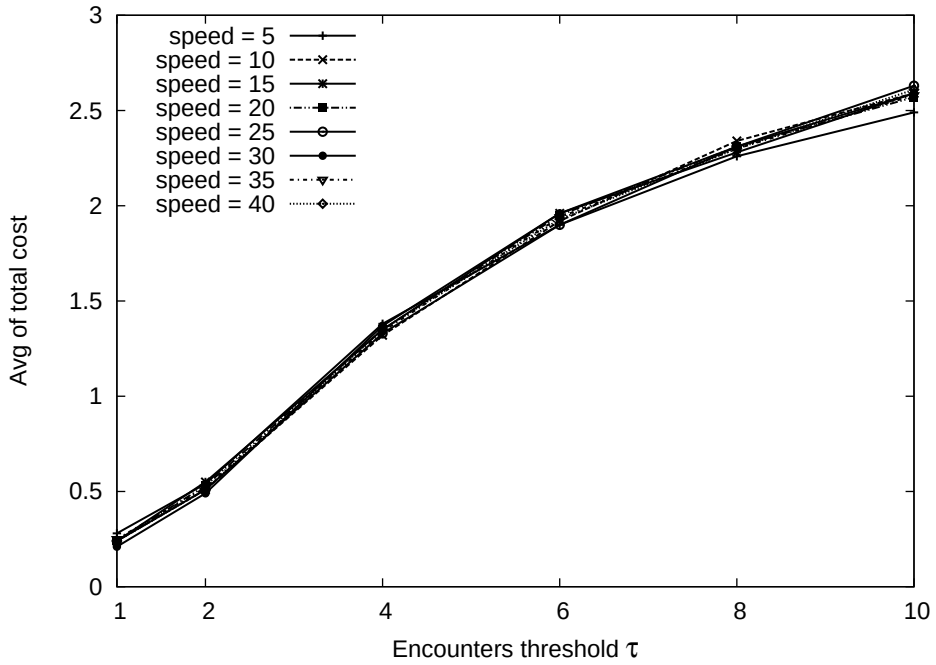

Figure 4.22: The average number of transmissions per a node vs the encounters threshold τ. Wireless range = 100m

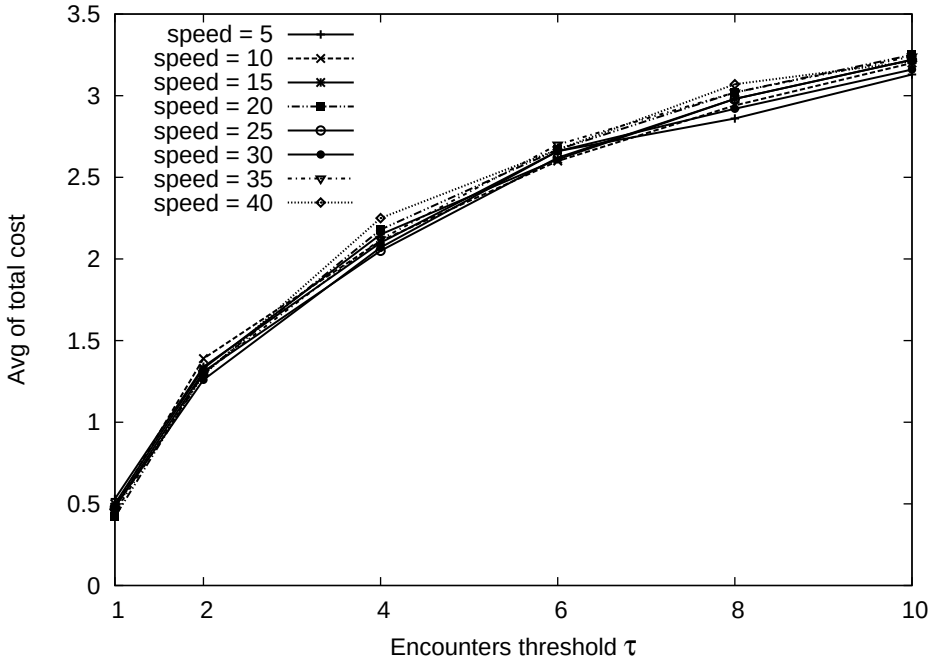

Figure 4.23: The average number of transmissions per a node vs the encounters threshold τ. Wireless range = 150m

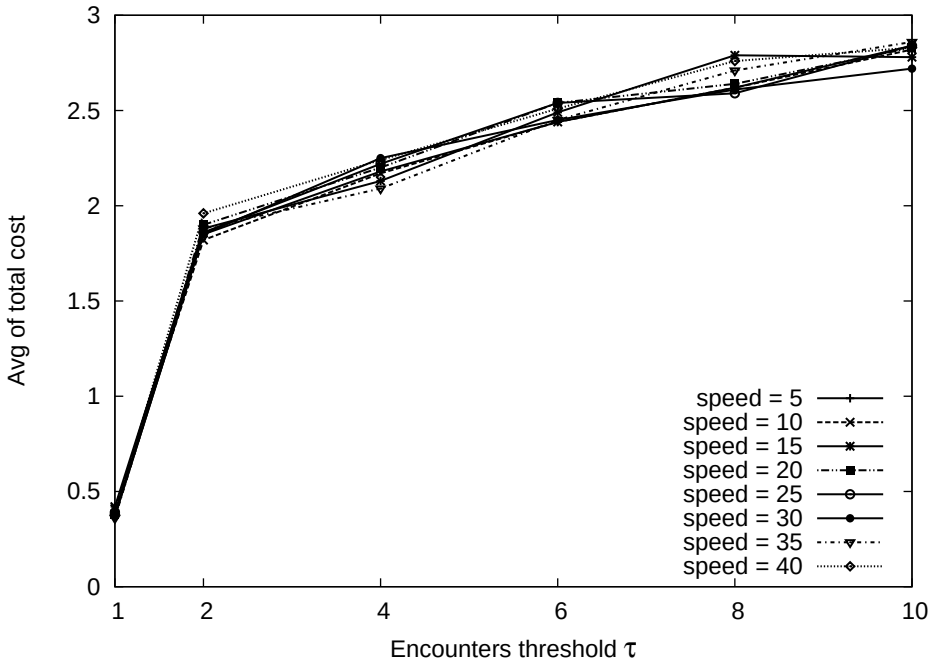

Figure 4.24: The average number of transmissions per a node vs the encounters threshold τ. Wireless range = 200m

Figures 4.25 - 4.27 show the average of group response time. It is observed (as expected) that low speeds incur longer delay because encounters are less frequent. This delay is, particularly, long when the speed is very low (5 m/s), increasing the speed after the latter one reduces the group response time substantially.

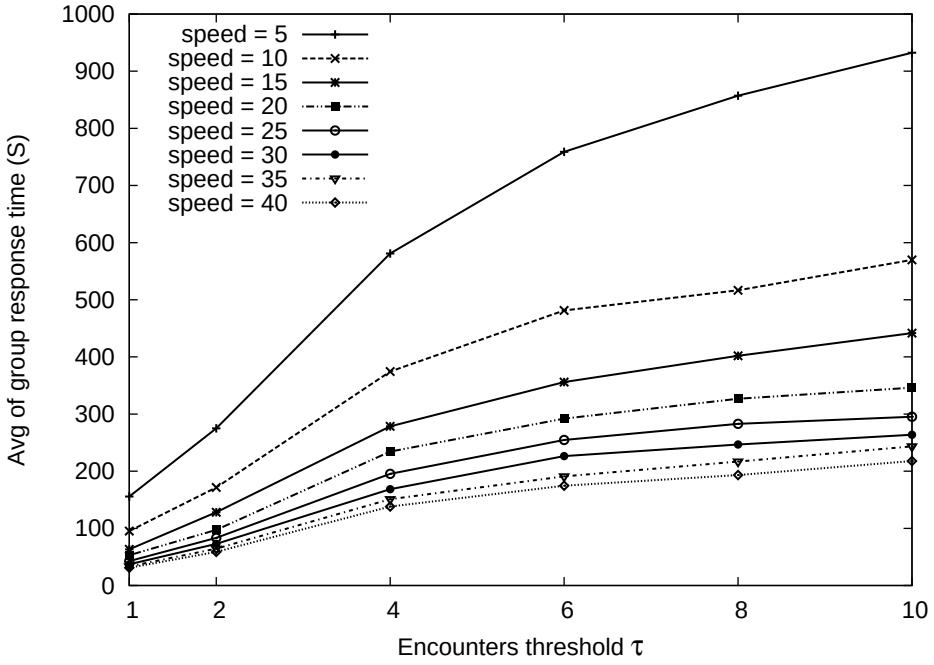

Figure 4.25: The group response time (second) vs the encounters threshold τ. Wireless range = 100m

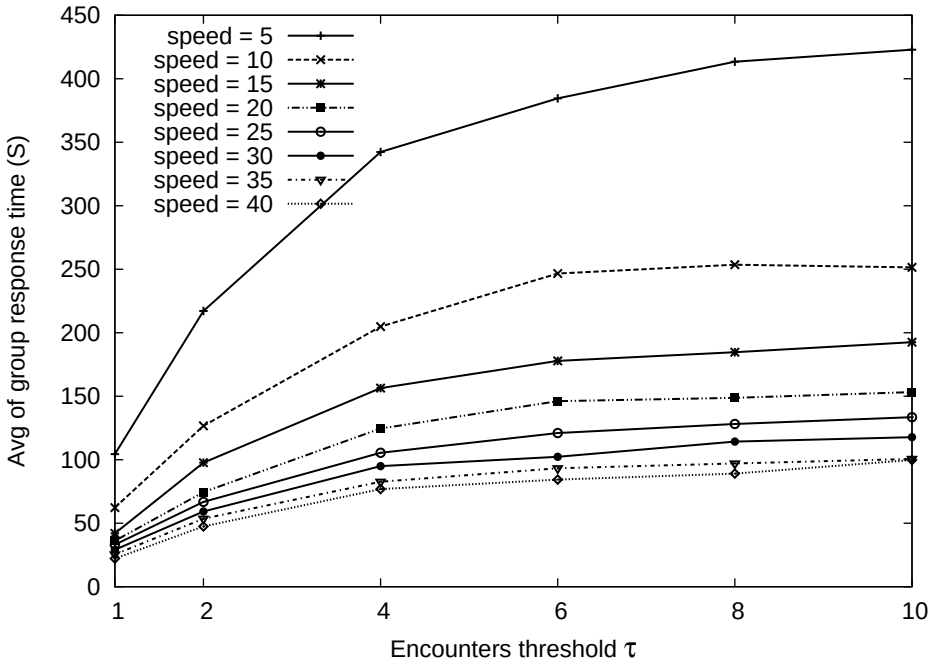

Figure 4.26: The group response time (second) vs the encounters threshold τ. Wireless range = 150m

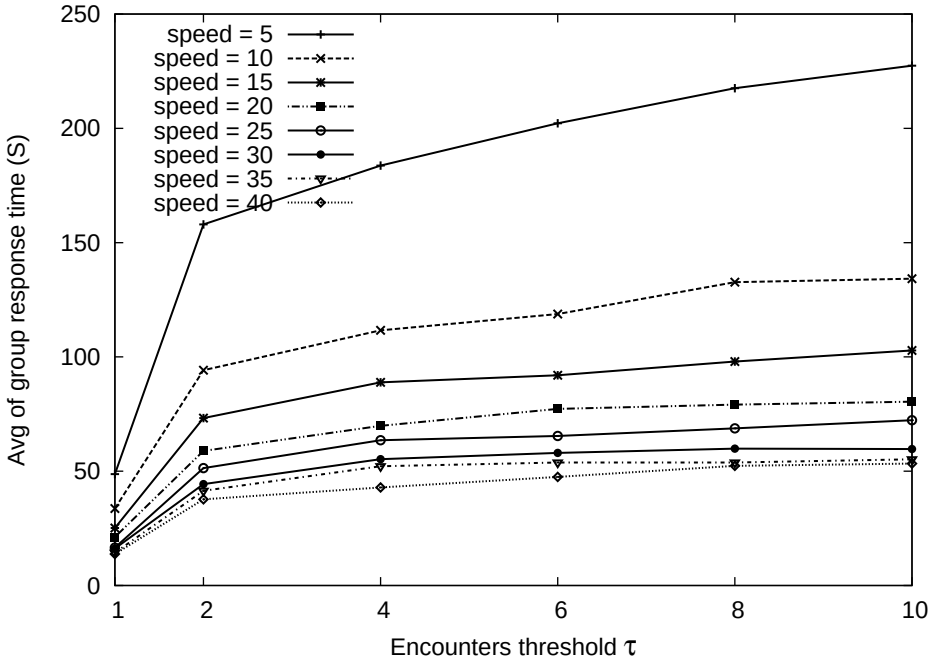

Figure 4.27: The group response time (second) vs the encounters threshold τ. Wireless range = 200m

4.4 Summary

In this chapter, the Encounter Gossip Multicast *EGM* protocol was derived from the Encounter Gossip *EG* broadcast protocol. The new derived protocol is a $\Diamond R$ protocol. This protocol is also a network topology-independent which makes it work without the need to build any routes. The latter property is preferable for MANETs where building and maintaining routes often pose extra overhead due to node mobility. *EGM* uses occasional flooding in addition to the group encounter based. The flooding, which is controlled by the encounters-to-flood threshold ϕ, is used to prevent the long delay which might be caused by the rare group encounters. However, the excessive use of flooding will impose extra overhead. Therefore, we introduce an optimization for *EGM* which uses an adaptive value for the threshold ϕ. The optimized *EGM* gives a good coverage with affordable cost and delay. So when we refer to *EGM* from any other chapters in this thesis, we mean the optimized *EGM*.

Chapter 5

Eventual Relinquishing/Quiescent Multicast Protocols

The last chapter has introduced the *EGM* protocol which is a $\diamondsuit R$ protocol. This protocol gives a good performance as was shown in simulations. However, *EGM* does not provide guarantee on the number of nodes that would certainly receive a multicast message. So this chapter will introduce our new $\diamondsuit RC$ and $\diamondsuit QC$, which both provide a delivery guarantee as was mentioned in the definition of these protocols.

As was explained, $\diamondsuit RC$ and $\diamondsuit QC$ use two underlying services. The first service is optional and it can be any $\diamondsuit R$ protocol. The second service provides an up-to-date view on the H-hop neighbourhood.

For the first service, we choose to implement $\diamondsuit RC$ and $\diamondsuit QC$ on top of the *EGM* protocol (as a $\diamondsuit R$ protocol). The *EGM* was chosen because (i) it was proved that this protocol performs well for a wide range of densities and node speeds, (ii) the provided coverage is not affected by network topology changes and (iii) it uses only one hop neigbourhood which can be maintained by just MAC level beacons.

We will also implement $\diamondsuit RC$ without using any $\diamondsuit R$ protocols. So we can compare the performances of this implementation and the implementation which uses *EGM* as a $\diamondsuit R$ protocol.

For the second service, the Neighbourhood Manager *NM* protocol will be presented. This protocol is used to build and maintain the H-hop ($H \geq 1$) neighbourhood list for group members.

So in the following sections, the $\diamondsuit RC$ and $\diamondsuit QC$ protocols will be presented and

we refer the reader to our work in [AE10] for the correctness discussions for these protocols.

5.1 $\diamondsuit RC$ Protocol Using *EGM* Protocol

As was mentioned, $\diamondsuit RC$ protocol satisfies two properties: Eventually relinquishing ($\diamondsuit R$) and the delivery guarantee of any multicast to $n - f$ members. So to fulfill these properties, $\diamondsuit RC$ uses the following sub protocols:

1. $\diamondsuit RC$-coordinator: This is the coordinator part which supervises the whole sending process of a message m until the required delivery guarantee is obtained (see Figure 5.1).

2. $\diamondsuit R$: This part is optional and it can be any protocol which satisfies the $\diamondsuit R$ property (e.g. the flooding protocol). In this section, we will use the Encounter Gossip Multicasting *EGM* protocol. This protocol will be instructed by $\diamondsuit RC$-Coordinator to multicast a message m (the first phase in Figure 5.1).

3. Delivery Guarantee Booster *DGB*: When *EGM* finishes sending m without attaining the required delivery guarantee, $\diamondsuit RC$-Coordinator needs to instruct *DGB* to start sending m. So *DGB* represents a booster part to help in gaining the required delivery guarantee (the second phase in Figure 5.1).

4. Neighborhood Manager *NM*: This protocol works to provide (i) 1-hop neighbours list, which includes member and non-member neighbours, to the *EGM* protocol, and (ii) H-hop group neighbours list ($H \geq 1$) to *DGB* and the routes to each neighbour when $H > 1$.

Figure 5.1 shows that each protocol ($\diamondsuit RC$-Coordinator, *EGM* and *DGB*) has four functions *send*, *deliver*, *kill* and *notify*. So when referring to these functions, we will add the protocol name to distinguish between them (e.g. *send* function of *EGM* protocol is denoted as *EGM_send*).

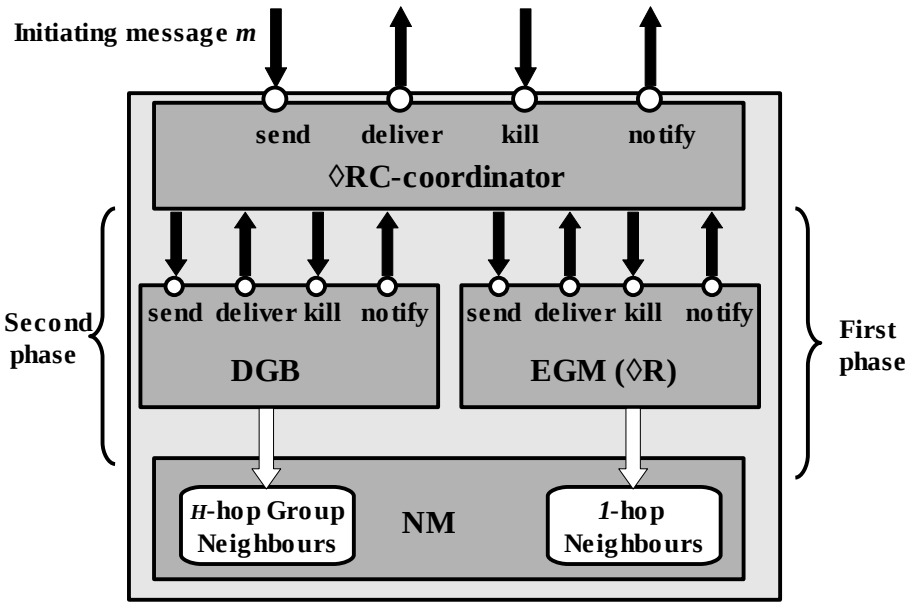

Figure 5.1: $\Diamond RC$ protocol sends messages in two consecutive phases

The major challenge in designing a $\Diamond RC$ protocol is in preventing a member N_i that received m from retaining m once the following property is met: at least $n - f$ members have received m. This property will be denoted as the realization of m. So member N_i, to be able to deduce this property, maintains knowledge regarding which other members have m. This knowledge is saved in a boolean vector, of n bits, which is denoted as $K_i(m)$. So $K_i(m)$ contains the values 0 or 1: (i) $K_i(m)[j] = 1$ means that node N_i knows that N_j has m, and (ii) $K_i(m)[j] = 0$ means that N_i does not know if N_j has m or not. Thus, $K_i(m)$ is maintained such that if it indicates that N_j has m, then it is certainly true; it needs not be true the other way round: when N_j does have m, $K_i(m)$ may indicate that N_j does not have m. Note also that $K_i(m)[j] = 1$ represents a specific knowledge of N_i about member N_j and m. So this knowledge does not change by time, therefore, number of 1s in $K_i(m)$ can only increase. Let $|K_i(m)|$ denote the total number of 1s in $K_i(m)$. Obviously, $K_i(m)$ is undefined if N_i had never 'seen' m, i.e., never received m.

In the following sections, we will describe the tasks of each sub protocol, in Figure 5.1, in details.

5.1.1 EGM protocol

This protocol works in the same way which was described in the last chapter except the transmission of the knowledge vector. So, with $\diamondsuit RU$ protocol, *EGM* has to include a copy of the knowledge vector $K_i(m)$ with any transmitted message m. As a result, members exchange the knowledge on the propagation of m by exchanging their knowledge vectors about m. This happens according to the following steps:

- A member N_i includes a copy of $K_i(m)$ with any transmitted message m.
- When a member N_j receives a message m:
    1. If N_j is receiving m for the first time, it stores a copy of the received vector $K(m)$ after changing its bit to 1, that is, $K_j(m)[j] = 1$. Moreover, N_j passes a copy of m accompanied with a copy of $K_j(m)$ to $\diamondsuit RC$-Coordinator by calling *EGM_deliver(m)*.
    2. If N_j already has a copy of m, it updates $K_j(m)$ according to the received $K(m)$ by adding the knowledge which is included in $K(m)$, that is, $K_j(m)[i]$ is set to 1, if either $K_j(m)[i]$ or $K(m)[i]$ is 1. If any changes occur to $K_j(m)$, N_j conveys these changes to $\diamondsuit RC$-coordinator by calling *EGM_deliver(m)*.

In this way members which are running *EGM* share the knowledge about m. In addition, this knowledge will be used by $\diamondsuit RC$-Coordinator which is informed about any changes to the knowledge vector $K(m)$ of a message m.

5.1.2 DGB Protocol

Describing this protocol requires adding some data structures which are stored by member nodes.

5.1.2.1 Local Data Structures

- In addition to the $K(m)$ vector, *DGB* needs each member to maintain, for every multicast message m, a boolean vector denoted as $KK(m)$. This vector stores the knowledge on $K(m)$. So each member N_i uses $KK_i(m)$ to maintain its knowledge on $K(m)$. That is, $KK_i(m)[j] = 1$ means that N_i knows that member node N_j has the same $K(m)$ as itself; otherwise, $KK_i(m)[j] = 0$. Note

that N_i has to reset $KK_i(m)$ every time $K(m)$ is changed because any changes to $K(m)$ affect the whole knowledge in $KK_i(m)$. So the number of 1s in $KK_i(m)$ might decrease. Moreover, when $KK_i(m)[j]$=1, N_i can only conclude that N_j had the same *K(m)* as itself at some time during the execution, because N_j may have changed its *K(m)* without N_i being aware of this change.

- H-hop group neighbourhood list ($Neigh_H$): Every member N_i maintains a list of all H-hop group neighbours $Neigh_{Hi}$ (or $Neigh_i$) which is provided and maintained by *NM* protocol.

- Routes list: For each neighbour N_j of a member N_i ($N_j \in Neigh_i$) a routes list ($Routes_j$) to this neighbour is provided by the *NM* protocol. Each entry of this list contains the ID of the next hop node towards N_j; the next hop might be the destination node N_j itself, or any non-member node.

- Message pool M: Each member N_i maintains a message pool M_i in which all received messages are saved.

- Each member N_i maintains a list L_i of 3-tuples for each received m: $< m.id, K_i(m), KK_i(m) >$.

- Knowledge packet $K\_pkt(m)$: A member node N_i can use this packet to send its knowledge vector about m. So this packet carries just the id of m and $K(m)$.

- Request packet *req(m)*: This packet can be sent by a member N_i to request m from another member. So it carries just the id of m.

- Realize packet *realize(m)*: When a member N_i knows that at least $n - f$ members (including N_i itself) have m, it realizes m (which means that N_i does not need to send m any more). N_i announces this realization by sending the realize packet (*realize(m)*) which contains only the id of m.

5.1.2.2 DGB Execution Steps

Figure 5.1 shows that *DGB* uses the H-hop group neighbours list which is provided by the *NM* protocol. So when *DGB* has any messages to send, it instructs *NM* to start working to find these neighbours.

This protocol has two threads that work concurrently to multicast a message m. Moreover, the related packets are not processed on their arrival but at discrete instances: each thread sleeps for a random interval on $(0, \beta)$ where β is a parameter; when it wakes up, it processes all packets delivered to it while it was asleep and then goes to sleep again. We show below the execution steps of these threads for a member N_i.

Thread 1.

For every message $m \in M_i$, which is not realized yet, this thread executes the following steps when it wakes up before it goes to sleep again:

1. For every member $N_j \in Neigh_i$ if $KK_i(m)[j] = 0$, then node N_i unicasts $K\_pkt(m)$ to N_j.

2. If it receives, during the last sleeping interval, a request packet *req(m)* from a neighbour N_j, it unicasts m to N_j accompanied with a copy of its $K(m)$.

3. When N_i receives $K\_pkt(m)$, it adds all the information from the received $K(m)$ to its $K_i(m)$. That is, if $K_i(m)[j] = 0$ and $K(m)[j] = 1$, then $K_i(m)[j]$ is set to 1. After any changes to $K_i(m)$, N_i updates $KK_i(m)$ accordingly.

4. N_i realizes m (i) if it receives *realize(m)*, or (ii) when $|K_i(m)| \geq n - f$.

5. If N_i realizes m, it unicasts *realize(m)* to each $N_j \in Neigh_i$, invokes *DGB_notify(m)* and deletes m from M_i.

Thread 2.

The following steps are executed when this thread wakes up:

1. If N_i receives a new message $m \notin M_i$:

   - It copies the received $K(m)$ to its $K_i(m)$ after changing $K_i(m)[i]$ to 1, and also creates the $KK_i(m)$ vector and sets all bits to 0 except $KK_i(m)[i] = 1$.
   - It stores a copy of m in M_i and $< m.id, K_i(m), KK_i(m) >$ in L_i.
   - It passes a copy of m to $\Diamond RC$-Coordinator by invoking *DGB_deliver(m)*.

2. When N_i receives any $K\_pkt(m)$, from another member N_j, for a message m which has not been received by N_i, it unicasts $req(m)$ to N_j.

3. If N_i receives $realize(m)$ for m, which has not been received, it ignores this packet.

4. When N_i receives m, *req(m)*, or $K\_pkt(m)$ from a member N_j for a realized m, it unicasts *realize(m)* to N_j.

Note that when $< m.id, K_i(m), KK_i(m) > \in L_i$ and $m \notin M_i$, this means that m is a realized message by N_i.

5.1.3 $\diamondsuit RC$-coordinator

$\diamondsuit RC$-Coordinator requires each member N_i to maintain a list $List_i$ of 2-tuples $< m.id, K_i(m) >$ for each received m. We will present the interface provided by $\diamondsuit RC$-Coordinator to the local application before we explain the tasks of $\diamondsuit RC$-Coordinator.

5.1.3.1 $\diamondsuit RC$-coordinator Interface

$\diamondsuit RC$-Coordinator provides the upper layer/protocol of a member N_i with the following functions (Figure 5.1):

- *send(m)*: When the upper protocol has a message m to send, it invokes *send(m)* to instruct $\diamondsuit RC$-Coordinator to send m.

- *deliver(m)*: When $\diamondsuit RC$-Coordinator receives a new message m, it invokes *deliver(m)* to pass m to the upper protocol.

- *kill(m)*: The upper protocol can call this function to instruct $\diamondsuit RC$-Coordinator to realize m, and so to stop handling m.

- *notify(m)*: When $\diamondsuit RC$-Coordinator realizes m, it informs the upper protocol about this realization by calling *notify(m)*.

5.1.3.2 $\diamondsuit RC$-coordinator Execution Steps

As was mentioned, $\diamondsuit RC$-Coordinator represents the coordinator for the whole $\diamondsuit RC$ protocol. So when a member N_i has a message m to send, it invokes $\diamondsuit RC\_send(m)$ to

pass m to $\diamondsuit RC$-Coordinator which handles this message according to the following steps:

1. It creates and initializes $K_i(m)$ as a vector of zeros and sets its own bit to 1, that is, $K_i(m)[i] = 1$. Then, N_i stores $< m.id, K_i(m) >$ in $List_i$.

2. It hands over m, accompanied with $K_i(m)$, to EGM which starts sending m to other members.

3. While EGM is sending m, $\diamondsuit RC$-Coordinator will be listening to any changes in the knowledge vector $K_i(m)$:

   a) If $\diamondsuit RC$-Coordinator receives any changes to $K_i(m)$ from EGM, it replaces $K_i(m)$ in $< m.id, K_i(m) >$ by the new received one.

   b) If $|K_i(m)| \geq n - f$: $\diamondsuit RC$-Coordinator realizes m, invokes *EGM_kill(m)* and calls $\diamondsuit RC$*_notify(m)*. Then $\diamondsuit RC$-Coordinator passes m to DGB, by calling *DGB_send (m,K(m))*, just to make DGB aware of the realization.

   c) When $\diamondsuit RC$-Coordinator gets notified by EGM which finishes sending m, it executes the next step.

4. $\diamondsuit RC$-Coordinator hands over m, accompanied with $K_i(m)$, to DGB which starts sending m (handing over m can be done by invoking *DGB_send (m,K(m))*).

5. When $\diamondsuit RC$-Coordinator gets notified by DGB, it invokes $\diamondsuit RC$*-Coordinator_notify(m)* to inform the upper protocol.

6. When $\diamondsuit RC$-Coordinator receives a new message m passed by EGM, it stores $< m.id, K_i(m) >$ in $List_i$, invokes $\diamondsuit RC$*_deliver(m)* and executes step 3.

7. When $\diamondsuit RC$-Coordinator receives a new message m passed by DGB, $\diamondsuit RC$-Coordinator just stores $< m.id, K_i(m) >$ in $List_i$ and invokes $\diamondsuit RC$*_deliver(m)*.

5.2 $\diamondsuit QC$ Protocol Using EGM Protocol

This protocol is similar to the $\diamondsuit RC$ protocol. So $\diamondsuit QC$ has the same structure of $\diamondsuit RC$ in Figure 5.1 after replacing the name of $\diamondsuit RC$-coordinator by $\diamondsuit QC$-coordinator. However, the main difference between $\diamondsuit QC$ and $\diamondsuit RC$ is that in $\diamondsuit QC$ messages are not relinquished upon realization. Therefore, $\diamondsuit QC$-coordinator and

DGB will be slightly different in this protocol while *EGM* and *NM* work exactly in the same way like $\diamondsuit RC$ protocol. So in the following sections, we will introduce *DGB* and $\diamondsuit QC$-coordinator.

5.2.1 DGB Protocol

Members here have to act in a different way, comparing with *DGB* in $\diamondsuit RC$, after the realization of a message m. This section will explain the differences in the behaviour of *DGB* between $\diamondsuit QC$ and $\diamondsuit RC$. These differences require some extra data structures which will be introduced later in this section.

DGB here needs to work for an extra time, beyond the realization of m, until reaching quiescence. That is, until each member is not required to unicast m or any packets regarding m. We will try to keep the cost incurred during the extra time to minimum. So we will add two actions which can help to reduce this cost:

1. *DGB* will use only 1-hop group neighbours after the realization of m by making $H = 1$. This will reduce the number of transmitted control packets because 1-hop group neighbours can be maintained with just MAC level beacons. So no control packets are needed for maintaining group neighbourhood.

2. *DGB* will use a new vector after the realization of m, instead of $KK(m)$, to deduce the need of unicasting control packets. Let us suppose that we use $KK(m)$ in the same way like $\diamondsuit RC$; In $\diamondsuit RC$, *DGB* uses the knowledge in $KK(m)$ to decide if a member needs to unicast $K\_pkt(m)$, but $KK(m)$ is reset whenever a change occurs to $K(m)$. This causes $KK(m)$ to take a long time to build the required knowledge for reaching quiescence because quiescence cannot be reached unless the number of 1s in $KK(m)$ is at least $n - f$. Therefore, a new vector of bits, denoted as $RV(m)$, will be used with *DGB* here. This vector will be introduced shortly.

5.2.1.1 Local Data Structures

DGB uses the same data structures used earlier with *DGB* in $\diamondsuit RC$ in addition to the following data structures:

- Another vector of bits, denoted as $RV(m)$, is needed to store the knowledge on the realization of m. So each member N_i maintains $RV_i(m)$ where $RV_i(m)[j] =$

1 means that N_i knows that N_j has realized m, while $RV_i(m)[j] = 0$ means that N_i does not know if N_j has realized m or not. Note that $RV_i(m)[j] = 1$ represents a certain knowledge of N_i about member N_j and m. So this knowledge does not change by time, therefore, number of 1s in $RV_i(m)$ does not decrease. The number of 1s in $RV_i(m)$ will be denoted as $|RV_i(m)|$. Moreover, $RV_i(m)$ is undefined if N_i has not realized m.

- The realize packet *realize(m)* contains, in addition to the id of m, a copy of the $RV(m)$ vector.

5.2.1.2 Execution Steps

DGB uses the H-hop ($H \geq 1$) group neighbours list until the realization of received messages. So when all messages in M_i for a member N_i are realized, N_i uses only 1-hop group neighbours list by setting H to 1.

DGB here works in a similar way like *DGB* in $\diamondsuit RC$. So this protocol uses the same two threads (Thread 1 and Thread 2) which were used in $\diamondsuit RC$ after applying some changes on each thread. In addition to these two threads, *DGB* here uses a new thread (Thread 3) for the realized messages. This new thread also operates on a sleep-and-act basis. Thus, this thread handles the received packets after a random sleep on $(0, \beta)$. In the following, we show the tasks carried out by Thread 3 as well as the changes on Thread 1 and Thread 2 for a member N_i. Note that when using any execution steps from $\diamondsuit RC$, $\diamondsuit RC$-coordinator is replaced by $\diamondsuit QC$-coordinator.

Thread 1.

Tasks 4 and 5 are changed as follows:

4. N_i realizes m when one of the following occurs: (i) N_i receives *realize(m)*, so N_i copies the received $RV(m)$ to its $RV_i(m)$ after changing $RV_i(m)[i]$ to 1. (ii) $|K_i(m)| \geq n - f$, in this case, N_i initialize $RV_i(m)$ with all bits being set to 0 except the i^{th} bit which is set to 1.

5. If N_i realizes m, it unicasts *realize(m)*, accompanied with a copy of $RV_i(m)$, to each $N_j \in Neigh_i$.

Thread 2.

Task 4 is deleted from this thread and task 3 is changed as follows:

3. If N_i receives *realize*(m) from a member N_j for m, which has not been received by N_i, it unicasts *req(m)* to N_j.

Thread 3.

For each message $m \in M_i$, which is realized, this thread carries out the following tasks when it wakes up before it goes to sleep again:

1. For every $N_j \in Neigh_i$ if $RV_i(m)[j] = 0$, then N_i unicasts *realize(m)* to the N_j accompanied with a copy of $RV_i(m)$.

2. If N_i receives, during the β interval, a request packet *req(m)* from a neighbour N_j, it transmits m to N_j accompanied with $K(m)$.

3. When N_i receives *realize(m)*, it adds all the knowledge from the received $RV(m)$ to its $RV_i(m)$. That is, if $RV_i(m)[j] = 0$ and $RV(m)[j] = 1$, then $K_i(m)[j]$ is set to 1.

4. N_i deletes m from M_i when $|RV_i(m)| = n$.

5.2.2 $\diamondsuit QC$-coordinator

This part works in the same way like $\diamondsuit RC$-coordinator after changing task b) in step 3 in $\diamondsuit RC$-coordinator to:

b) If $|K_i(m)| \geq n - f$: $\diamondsuit QC$-Coordinator invokes *EGM_kill(m)* and hands over m accompanied with $K(m)$ to *DGB*.

5.3 $\diamondsuit RC$ protocol Without Using $\diamondsuit R$ Service

In the last section, $\diamondsuit RC$ was implemented on top of the *EGM* protocol as an optional $\diamondsuit R$ service. In this section, $\diamondsuit RC$ will be implemented without the support of the $\diamondsuit R$ service. The purpose of this implementation is to study the effect of using this service with $\diamondsuit RC$ protocol.

The structure of this protocol is shown in Figure 5.2. This structure shows that this protocol uses $\diamondsuit RC$-coordinator, *DGB* and *NM*:

- $\diamondsuit RC$-coordinator works in the same way like $\diamondsuit RC$-coordinator in the last implementation of $\diamondsuit RC$ (sub section 5.1.3.2) after deleting steps 2, 3 and 6 which are related to using *EGM*.
- *DGB* here works exactly in the same way like *DGB* in the last implementation of $\diamondsuit RC$ (sub section 5.1.2.2).
- *NM* provides *DGB* with H-hop group neighbourhood list.

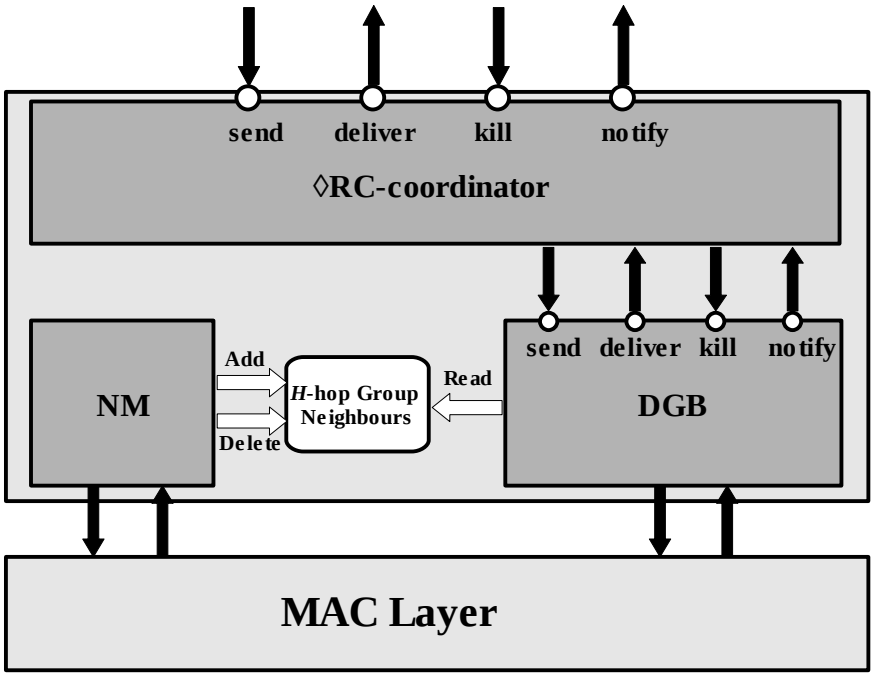

Figure 5.2: $\diamondsuit RC$ protocol without using $\diamondsuit R$ service

5.4 Neighbourhood Manager NM Protocol

This section will introduce the Neighbourhood Manager *NM* protocol which is used to build and maintain (i) H-hop ($H \geq 1$) group neighbourhood list and routes to each group neighbour (when $H > 1$), or (ii) 1-hop neighbourhood list ($H = 1$) which includes members and non-members. The 1-hop neighbourhood list and 1-hop group neighbourhood list do not need maintaining any routes. Moreover, as was explained before, these lists can be maintained with just MAC level beacons. So no control packets are needed for maintaining these lists. This section will concentrate on the

cases when $H > 1$ where group neighbourhood list and routes to each neighbour need to be maintained.

NM is a mesh-based protocol which means that *NM* builds and maitains routes by buidling a mesh that connects group neighbours. So *NM*, as a mesh-based protocol, provides a high delivery ratio because it creates redundant routes. This enables this protocol to work even when some routes break due to node mobility. Moreover, *NM* is on-demand (reactive) protocol which means that it constructs and maintains routes on demand (only when they are needed). This reduces the overhead which is caused by building these routes.

NM protocol's mission is only to build and maintain routes between members, so it has to be supplemented with a multicast protocol. For example, This protocol is used in Figures 5.1 and 5.2 to build and maintain the H-hop neighbourhood list. The built neighbourhood list and routes are provided to the supplemented protocols on demand.

5.4.1 Overview of the H-hop Group Neighbourhood

When a member N_i has some other members inside its wireless range (which are considered neighbours of N_i), it can communicate with them without the need to build any routes. This kind of neighbourhood can be called one hop neighbourhood. However, in many scenarios this kind of communications between members cannot be attained due to node mobility and low group density (which is the number of group members $|G|$ to the total number of nodes in the system $|\mathcal{S}|$). Therefore, H-hop group neighbourhood can be built to attain higher connectivity between members. So according to the H-hop group neighbourhood, members can have H-hop group neighbours where H can be any integer greater than 0. Building this neighbourhood requires defining the following terms and data structures:

- Group hello packet *Ghello*: Group members can advertise their existence to other members in the area by sending *Ghello* packets. So when a member N_i needs to discover the H-hop group neighbours, it sends *Ghello* packet which can be forwarded $H-1$ times, by non-member nodes, until it reaches another member N_j. This packet carries the following information:

  1. The multicast group ID (*MGID*).

2. Time to live TTL: This variable refers to the number of times the *Ghello* packet can be transmitted/forwarded. So every time a node transmits/forwards the *Ghello*, it reduces TTL by 1, nodes stop forwarding the Ghello packet when TTL=0. TTL is always initiated to H.

3. The generators ID list ($GIDL$): This list contains the IDs of all nodes which have transmitted/forwarded the group hello packet where GIDL[0] carries the ID of the member N_i which created this packet.

- Routes or forwarding table (denoted as *Routes*): Every node in the system has a forwarding table which is used to save the routes information between members.

- Route Entry RE: The forwarding table contains the routes entries which describe routes. So each entry inside this table represents one route and it involves the following data:

  1. The route destination ID ($RDID$): It represents the ID of the member which is the final destination of this route.

  2. The route next hop ID ($RNID$): It is the ID of the next node on the route towards the destination.

  3. The ID of the multicast group ($RMGID$).

  4. The route creation time RT: This is the time when this entry was generated.

  5. The route priority RP: It decides the priority of the route. This variable can be used when there are more than one route to the same destination. So the sender can choose the route with the greatest priority.

- H-hop group neighbourhood list $Neigh_H$: It stores the group neighbours for a member N_i. So each member needs to have a neighbourhood list for its neighbours.

5.4.2 NM Description

This protocol works according to the following steps:

- When a member N_i needs to find routes to its H-hop neighbours, this member is marked as an active member. This activation means that N_i periodically (every μ time) has to broadcast *Ghello* packets to other members. So this node starts by generating a *Ghello* packet and setting the content of this packet as follows:

  1. *MGID*: It is set to the ID of the multicast group which the member N_i belongs to.
  2. *TTL*: It is set to H.
  3. GIDL: This list is set to empty and then N_i is added to it.

- N_i transmits the generated *Ghello* after reducing *TTL* by 1.

- When any node receives the *Ghello* packet, it reads the content of this packet (*MGID*, *TTL*, and *GIDL*) and behaves as follows:

  1. It uses the content of the received *Ghello* to create a route entry RE= $\{RDID, RNID, RMGID, RP, RT\}$ where ($RDID$= $GIDL[0]$, $RNID$=$GIDL[H-TTL-1]$, $RMGID$=$MGID$, RP=TTL, RT=current time). Then, it checks its routes table: (i) If it finds any entry $RE_1 \in$ *Routes* where ($RDID_1$=$RDID$, $RNID_1$=$RNID$ and $RMGID_1$= $RMGID$), it replaces RE_1 by RE in *Routes*, otherwise, (ii) it adds the route entry RE to its *Routes*.
  2. If it is a member node (denoated as N_j), it adds the received $GIDL[0]$ to $Neigh_j$. Moreover, if N_j is not an active member, it creates and transmits a *Ghello* packet for one time. This enables other members to discover N_j when they receive this *Ghello*.
  3. If it is a non-member node (denoted as X_j) and the received *TTL* (in the received *Ghello*) is greater than 0, it adds its ID to the $GIDL$ in the received *Ghello*, reduces *TTL* by 1 and then transmits this *Ghello* again.

- When a node needs to use a route to a destination member N_i, it searches in its *Routes* to find out if this table has a route to N_i; When *Routes* contains more than one route, the route with the greatest route priority RP is chosen.

- a route entry $RE \in Routes$ of any node is deleted when it ages. A route entry ages when the node does not receive any *Ghello* from the same *RDID* of *RE* during an interval of time. This latter interval is a parameter and it needs to be chosen carefully.

- A member N_i stops sending *Ghello* packets when it gets inactivated by the upper protocol.

5.4.3 Example of 2-hop Group Neighbourhood

Figure 5.3 shows an example for 2-hop neighbourhood between group members. The system contains 10 nodes of which 3 are group members N_i, N_j and N_k. Figure 5.3 shows that every group member has the following neighbours and routes to each neighbour:

Member N_i:

The group neighbours $Neigh_i = \{N_j\}$ with two routes to this neighbour:

- $N_i \rightarrow X_i \rightarrow N_j$

- $N_i \rightarrow X_j \rightarrow N_j$

Member N_j:

$Neigh_j = \{N_i, N_k\}$ with two routes to each neighbour:

- $N_j \rightarrow X_i \rightarrow N_i$

- $N_j \rightarrow X_j \rightarrow N_i$

- $N_j \rightarrow X_k \rightarrow N_k$

- $N_j \rightarrow X_l \rightarrow N_k$

Member N_k:

$Neigh_k = \{N_j\}$ and the routes to this neighbour are:

- $N_k \rightarrow X_k \rightarrow N_j$

- $N_k \rightarrow X_l \rightarrow N_j$

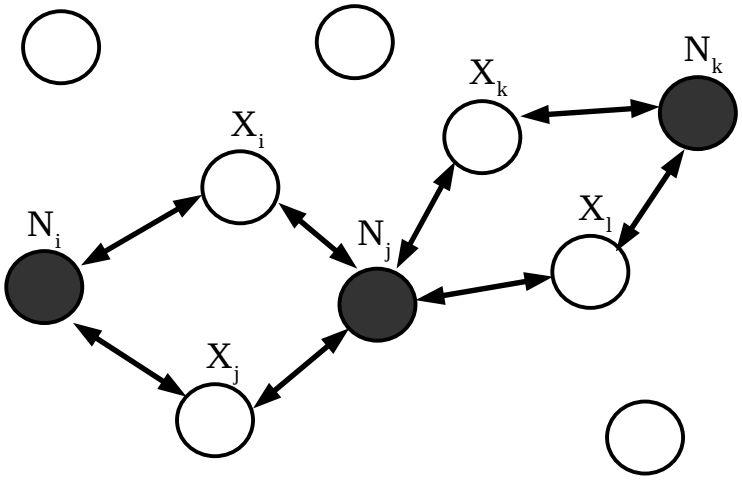

Figure 5.3: The 2-hop neighbourhood between group members. The arrowa show the routes between the neighbours

The routes between the group neighbours maintain the property of robustness. So when some changes occur to the network topology, nodes can still communicate by using the redundant routes. Figure 5.4 shows the same scenario in Figure 5.3 when node N_j moves to a new position. In the latter case, member nodes can still communicate with each others by using the following routes:

- $N_i \rightarrow X_j \rightarrow N_j$
- $N_j \rightarrow X_j \rightarrow N_i$
- $N_j \rightarrow X_l \rightarrow N_k$
- $N_k \rightarrow X_l \rightarrow N_j$

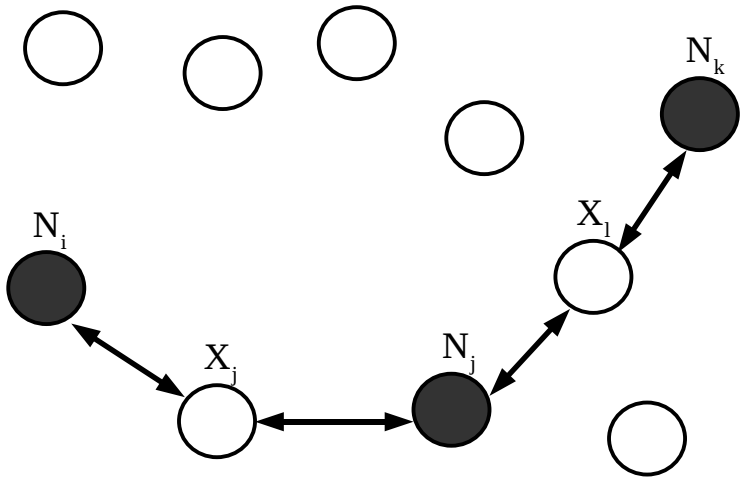

Figure 5.4: The 2-hop neighbourhood after some members move away

5.5 Comparison Between $\diamondsuit Q$ and $\diamondsuit QC$

In chapter 2, we described a quiescent implementation of the reliable broadcast protocol which was developed in [ACT00]. This section will compare between the described quiescent implementation $\diamondsuit Q$ and our quiescent protocol $\diamondsuit QC$ which was described in details earlier in this chapter.

The quiescent implementation in [ACT00] uses the HB failure detector. HB enables the quiescent protocols to distinguish crashed nodes for the purpose of preventing any node to send messages to these crashed nodes. This failure detector requires nodes to periodically send a heartbeat and store a vector of counters (one counter for each neighbour N_j of node N_i). So when a node N_i receives a heartbeat from a neighbour N_j, it increases the counter of N_j, so N_i assumes that N_j is crashed when the counter of N_j stops increasing. In addition to the HB, the quiescent implementation in [ACT00] requires a node N_i to store, for each message m, a knowledge vector $has_i[m]$ which contains the IDs of nodes that it is known to N_i that these nodes have m. So by using this vector, a node N_i can avoid sending m to any neighbour that has m.

In our $\diamondsuit QC$ protocol, each member periodically sends *Ghello* packet to other

members. This packet enables members to build the group neighbourhood list. Moreover, $\diamondsuit QC$ uses the following data structures:

- Each member N_i stores, for each message m, a knowledge vector $K_i(m)$ regarding which other members have m.
- For each message m, a member N_i maintains another knowledge vector $KK_i(m)$ which stores the knowledge on $K(m)$. After the realization of m, $KK_i(m)$ is replaced by another vector $RV_i(m)$ which stores the knowledge on the realization of m.

So we notice that the quiescent implementation of the reliable broadcast and our quiescent protocol $\diamondsuit QC$ are both similar in their ability of preventing nodes from sending messages to the crashed nodes. That is, using the heartbeat in the former and the *Ghello* in the latter. Moreover, the two protocols use knowledge vectors ($has[m]$ and $K(m)$) to store the knowledge regarding which nodes/members have m. On the other hand, a member N_i in $\diamondsuit QC$ uses, for each message m, an extra vector $KK_i(m)$ which stores information as whether other members have the same $K(m)$ as N_i. So when N_i has a group neighbour N_j which is deemed not to have the same $K(m)$ as $K_i(m)$, then N_i sends only $K_i(m)$ to N_j. If N_j has not received m, it learns of the existence of m and a request for m to be sent. Thus, N_i sends m to a neighbour only on request. After the realization of m, N_i deletes $KK_i(m)$ and starts using $RV_i(m)$ to store the knowledge on the realization of m. This vector is used in similar way, when N_i has a group neighbour N_j which is deemed not to have realized m, then N_i sends $RV_i(m)$ to N_j. Thus, N_i does not send any packets if all its neighbours are known to have realized m.

5.6 Required Duration of Node Connectivity

The system satisfies the liveness condition (3.2) which was introduced in Chapter 3. **Claim 1**: Supposing that $\diamondsuit RC$ and $\diamondsuit QC$ use β as their sleep interval and *NM* uses μ as its *Ghello* interval; If $B > (2\mu + 3\beta + 5H\delta)$, then a *(B,H)-Connectivity* is long enough to support the sequence of information exchange triggered by the $\diamondsuit RC$ and $\diamondsuit QC$ protocols.

Suppose that N_i has received m and N_j has not, when *(B,H)-Connectivity* between them begins at time, say, t. The following sequence of events can occur, each

with the worst possible delay:

1. N_i emits *Ghello* at $t + \mu$ which reaches N_j at $(t + \mu + H\delta)$.

2. N_j responds by emitting *Ghello* at $(t + 2\mu + H\delta)$ which reaches N_i at $t + 2\mu + 2H\delta$, ensuring the presence of N_j in $Neigh_i$.

3. Thread 1 of N_i wakes-up at $(t + 2\mu + \beta + 2H\delta)$ and unicasts a $K\_pkt(m)$ which reaches N_j by $(t + 2\mu + \beta + 3H\delta)$.

4. Thread 2 of N_j processes the $K\_pkt(m)$ from N_i at $(t + 2\mu + 2\beta + 3H\delta)$ and unicasts a $req(m)$ which reaches N_i at $(t + 2\mu + 2\beta + 4H\delta)$.

5. Thread 1 of N_i responds to $req(m)$ at $(t + 2\mu + 3\beta + 4H\delta)$ and m arrives at N_j at $(t + 2\mu + 3\beta + 5H\delta) < t + B$.

5.7 Performance Study

The same simulation parameters listed in Table 4.1 are used here. 50 mobile nodes of $\mathcal{S}$ were randomly placed in a fixed size terrain of 1000m x 1000m. The multicast group G contains 10 nodes ($|G| = 10$) chosen randomly at the start of each run and they do not leave the group during this run. Also, at the start, three members are randomly chosen for crashing ($f = 3$) at randomly chosen moments.

The network density (as was shown in Table 4.1) was varied by varying the nodes' wireless range as 100, 150 and 200 meters, resulting in the density values 1.6, 3.5 and 6.3 respectively.

Each simulation, for a particular set of parameters, was repeated 1000 times using distinct random seeds. So, each point in the graphs we present is the average on measurements taken over 1000 runs. Moreover, each run started after 1000 seconds of node movement to avoid any initial bias in node placement.

The value of β in $\diamondsuit QC$ and $\diamondsuit RC$ protocols was varied as: 2 and 5 seconds. No significant difference in performance was observed and the results presented here use $\beta = 5$.

The value of H in $Neigh_H$ is 2, if a node has an unrealized multicast, whether it is $\diamondsuit QC$ or $\diamondsuit RC$; H is set to 1, once all on-going multicasts are realized. For $H > 1$, members send a *Ghello* packet every 2 seconds interval to update $Neigh_H$. Each member allows 2 losses of *Ghello* packet before deleting a group neighbour.

So membership of $Neigh_H$ and a route information are lost, unless renewed within $3 \times 2 = 6$ seconds. Constructing $Neigh_H$, for $H = 1$, takes advantage of MAC level beacons being periodically sent by mobile devices to announce their presence to their immediate (1-hop) neighbours. So no extra control packets are required in this case.

EGM is using the value 6 for τ ($\tau = 6$) and the values for ϕ are the same which are shown in Table 4.2.

5.7.1 $\diamondsuit RC$ Protocol Using EGM Protocol

The protocol starts a run by randomly choosing an operative member to send one message m. This run continues until all operative members, which have received m, realize m. Note that the crashing members are excluded when choosing the member which initiates m.

We measured 3 performance metrics for 3 different densities, and 8 different maximum node speeds. They are presented as: relinquishing time, average of sent data packets and average of sent control packets. The first one is categorized as time overhead and the other two as packet overhead.

- Relinquishing time for a message m: The time taken until the last operative member, which has m, realizes m starting from the moment when m is initiated.

- Average of sent data/control packets for multicasting m: The total number of data/control packets transmitted by all nodes until the last member realizes m, divided by $|\mathcal{S}| = 50$. A data packet refers to any m containing data passed to $\diamondsuit RC$ by the local application, while all other packets are counted as control packets (which do not however include MAC level beacons).

Figure 5.5 shows that when the node density is lower, the relinquishing time is always longer. The relinquishing time reduces when the node speed increases (up to a threshold). This is because (i) $\diamondsuit RC$ is using *EGM* protocol which shows the same trend when increasing the speed (see Figures 4.25 - 4.27), and (ii) as the speed increases $(B, H \geq 2)$-*Connectivity* is formed quickly; beyond the threshold, connectivity does not last for the required B duration due to fast node movement.

Doubling the density value reduces relinquishing time by more than half at all node speeds.

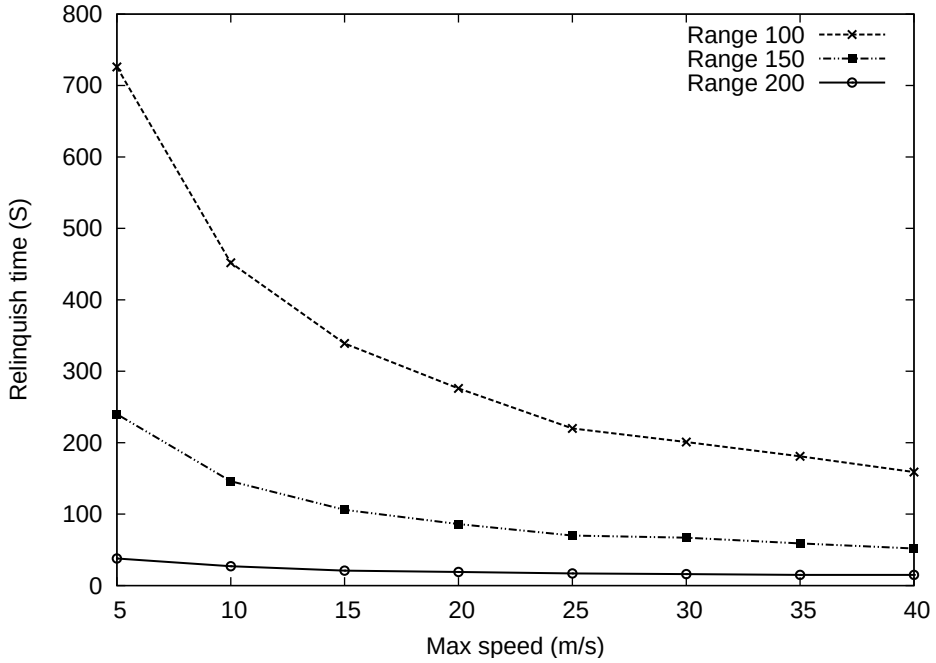

Figure 5.5: The relinquishing time (seconds) vs the max speed (m/s)

Figure 5.6 shows that the node speed has no important effect on the number of transmitted data packets. This is because the node speed has no effect on the transmitted packets in *EGM* (see Figures 4.22 - 4.24). It is also observed that the highest density (wireless range=200m) shows the smallest number of transmitted data packets. The reason for this is that at this density, members realize the message m quickly (Figure 5.5) and so *EGM* is ordered to stop sending m quickly (upon realizing m). Decreasing the density shows that higher density (wireless range=150m) incurs more transmitted data packets than lower density (wireless range=100m). This is explained by the effect of *EGM* which shows the same trend at these densities (see Figures 4.22 and 4.23).

Figure 5.7 shows that the relinquishing time has a big influence on the number of transmitted control packets per a multicast message. Indeed, the number of transmitted control packets shows a similar trend as the relinquishing time (Figure 5.5). This suggests that the longer it takes to realize the message, the more control packets are transmitted. Figure 5.8 shows the average number of transmitted *Ghello* packets, by *NM*, per node. This Figure shows the same trend as Figure 5.7. Moreover, when comparing between 5.7 and 5.8, we notice that the majority of transmitted control packets in 5.7 are actually *Ghello* packets. In general, more

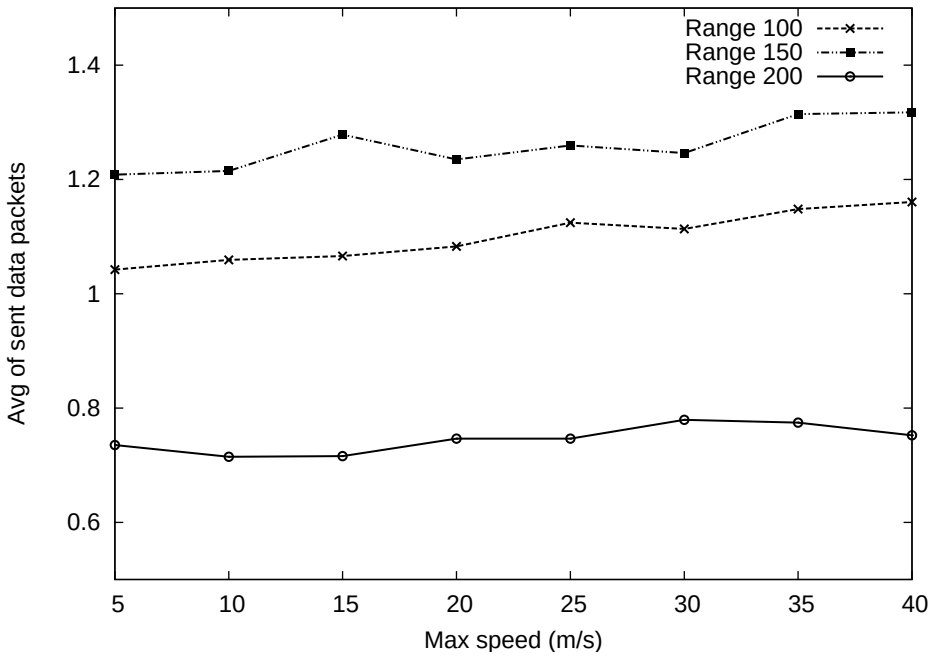

Figure 5.6: The average of sent data packets per node vs the max speed (m/s)

Ghello packets will be needed when the message takes longer time to be realized (longer relinquishing time). This explains the link between the trends in 5.5 and 5.7.

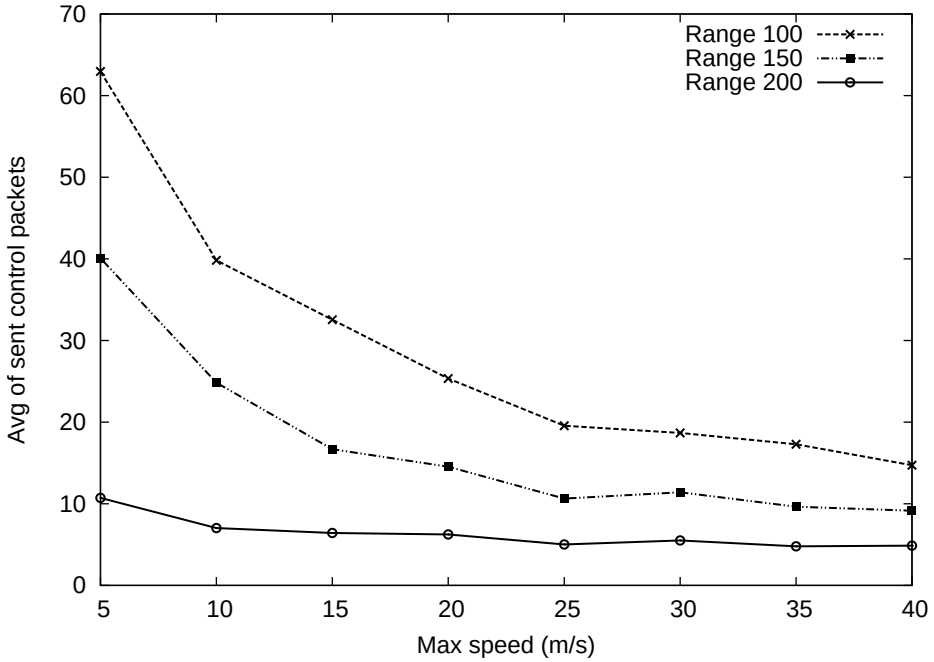

Figure 5.7: The average of sent control packets per node vs the max speed (m/s)

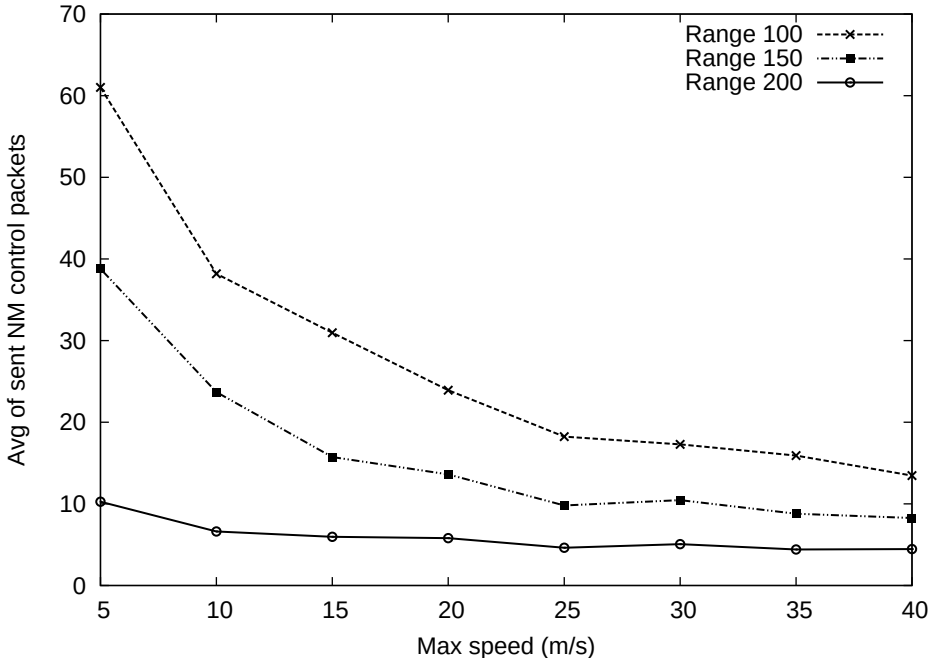

Figure 5.8: The average number of transmitted NM control packets per node vs the max speed (m/s)

5.7.2 $\Diamond QC$ Protocol Using EGM Protocol

We use exactly the same parameters used in the last section. The protocol starts by randomly choosing a member which sends one message, but the protocol here continues until all operative members reach the total quiescence after receiving m. However, in some cases reaching the total quiescence requires a long simulation time, in particular, with low density and slow movement. So to avoid the big delay in performing our experiments, we have set a maximum delay time MD for members. That is, when all members, in a run, do not perform any unicasting activities for a duration d ($d \geq MD$), this run is terminated. The value of MD used in our simulations is 500 seconds. In fact, the terminated cases were rarely encountered (less than 1%) during the whole 1000 runs for each set of parameters.

We measured only 2 performance metrics: Quiescence time (time overhead) and the total of sent control packets (packet overhead).

- Quiescence time for a message m: The time taken until all members become totally quiescent. The latter occurs when all operative members have identical $RV(m)$. The timing starts from the moment when m is initiated.

- Total number of sent control packets: The total number of control packets

transmitted by all nodes between the moment when last member realizes m until members become totally quiescent.

Figure 5.9 shows that the quiescence time has a similar trend as the relinquishing time in Figure 5.5. It is shown that members take some extra time to become quiescent comparing with the relinquishing time (as expected). This extra time is at its longest when density and maximum speed are at their lowest. Increasing density reduces the extra time incurred before quiescence. This extra time becomes almost negligible at the highest density (wireless range=200m).

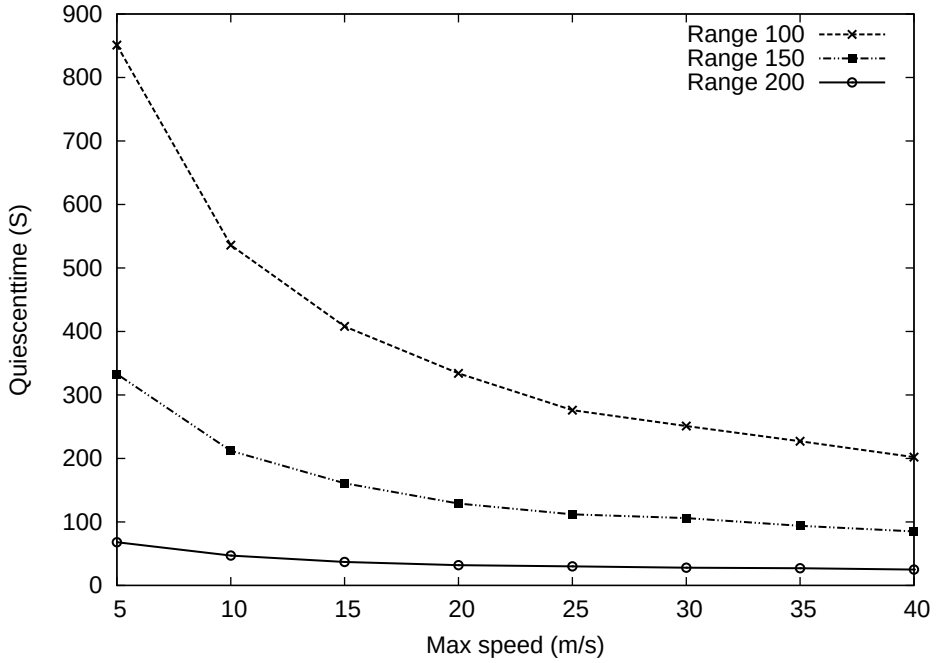

Figure 5.9: The quiescence time (seconds) vs the max speed (m/s)

Figure 5.10 shows the total number of control packets transmitted by all nodes starting from the moment when the last operative member, which has m, realizes m until members become totally quiescent. It is shown that only few numbers of control packets are sent in all densities. These numbers can be easily ignored; the maximum is 7 packets by 50 nodes. Considering that members after realizing m use only MAC level beacons to maintain the group neighbourhood list.

Note that there are no extra transmitted data packets after the last operative member realizes m. That is, considering that all data packets transmitted after the first member realizes m until the last operative member does are counted in Figure 5.6.

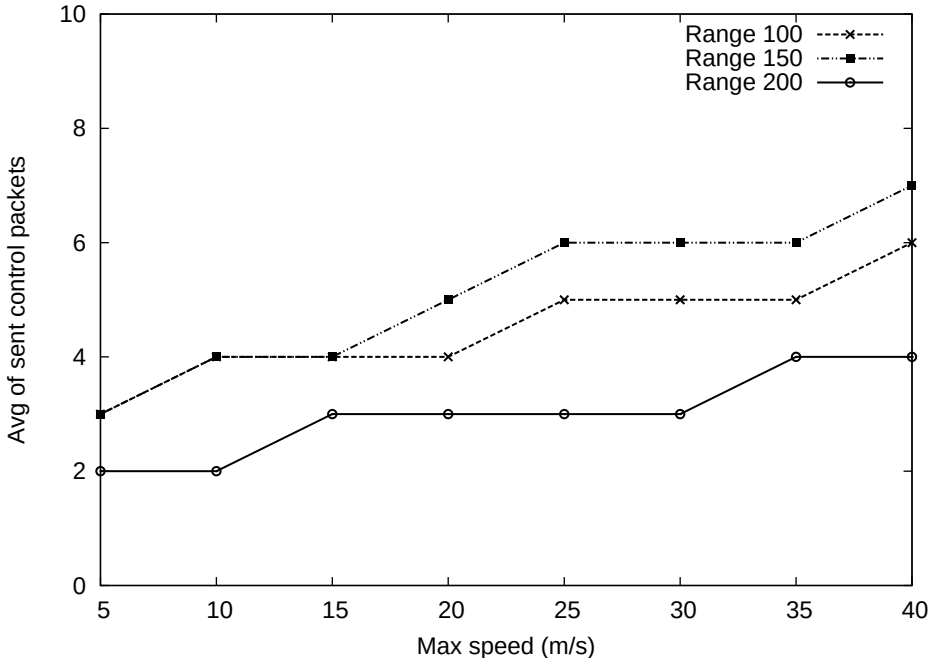

Figure 5.10: The total number of sent control packets vs the max speed (m/s)

5.7.3 $\diamondsuit RC$ protocol Without Using $\diamondsuit R$ Service

The simulation parameters used here are the same like the ones which used for $\diamondsuit RC$ in the last section. The only difference here is that *EGM* protocol is not used. So all parameters which are related to *EGM* (values of τ and ϕ) are ignored here. In the same way like in last section, $\diamondsuit RC$ starts a run by randomly choosing an operative member to send one message m. This run continues until all operative members, which have received m, realize m.

We measured here the same 3 performance metrics which measured for $\diamondsuit RC$ in the last section; relinquishing time, average of sent data packets and average of sent control packets. The only difference here is that the average of sent data packets refers to the total number of transmitted data packets by all nodes, so here this number is not divided by $|\mathcal{S}| = 50$.

Figure 5.11 shows that the relinquishing time has the same trend like the relinquishing time of $\diamondsuit RC$ (Figure 5.5) in the last section; lower densities incur longer time to relinquish the message. Moreover, increasing the node speed (up to a threshold) reduces the relinquishing time. It is also observed, when comparing Figures 5.11 and 5.5, that $\diamondsuit RC$ here takes shorter time to relinquish. This is because $\diamondsuit RC$ in this section sends messages only using *DGB* which depends on 2-hop members connectivity while $\diamondsuit RC$ in the last section uses *EGM* and *DGB* subsequently; *EGM*

depends on, only, 1-hop connectivity. This makes $\diamondsuit RC$ in the last section incur more delay before relinquishing the message in particular with lower densities due to lower 1-hop members connectivity.

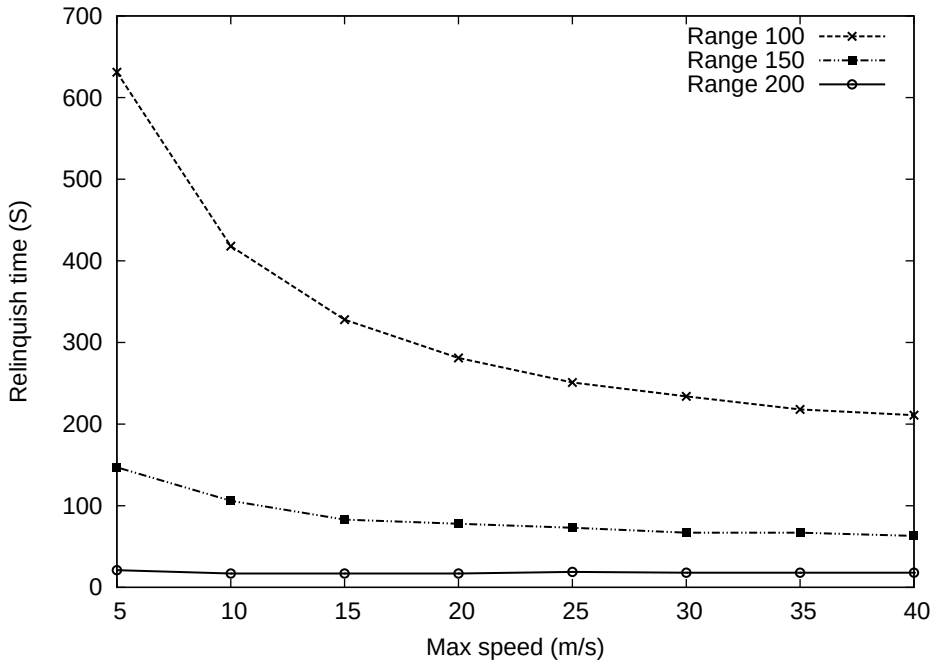

Figure 5.11: The relinquishing time (seconds) vs the max speed (m/s)

Figure 5.12 shows the total number of data transmissions by all nodes. It is shown that this number is quite small, in all densities, because $\diamondsuit RC$ in this section depends on DGB which unicasts messages only when they are requested. So a member N_i, which has not received m, will request m only once unless this message is lost after it is unicasted to it. It is also observed that the node density and the node maximum speed have very small effect on the number of transmissions. The two higher densities incur almost one extra transmission. This is may be due to faster connectivity, with higher densities, which enables some crashing nodes to request m before they go to crash.

Figure 5.13 shows that this protocol ($\diamondsuit RC$ in this section) transmits more control packets when the relinquishing time is longer (as expected). This can be observed when comparing 5.13 and 5.11 which both show the same trend. We will have more interest in comparing this protocol with $\diamondsuit RC$ in the last section in terms of the number of transmitted control packests. That is, by comparing between Figures 5.13 and 5.7. These figures show that this protocol incurs higher number of transmitted control packets in all densities. In particular, in the lowest and highest densities where (i) in the lowest density each node transmits almost 20 extra control packets

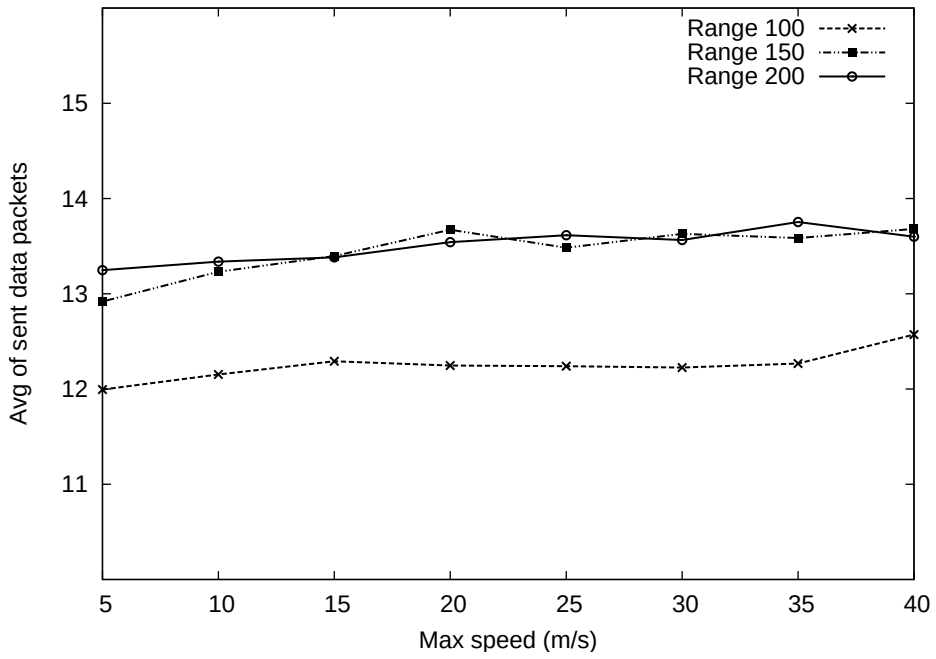

Figure 5.12: The average of sent data packets per node vs the max speed (m/s)

in all speeds, and (ii) in the highest density each node transmits more than double the number of transmitted control packets in $\diamondsuit RC$ in the last section (in all speeds). Note that this protocol transmits this higher number of control packets despite the fact that it incurs shorter delay (relinquishing time) than $\diamondsuit RC$ in the last section. This is because this protocol depends only on DGB which requires 2-hop group neighbourhood while $\diamondsuit RC$ in the last section depends on EGM and DGB; EGM, unlike DGB, requires only 1-hop neighbourhood which is obtained using MAC level beacons.

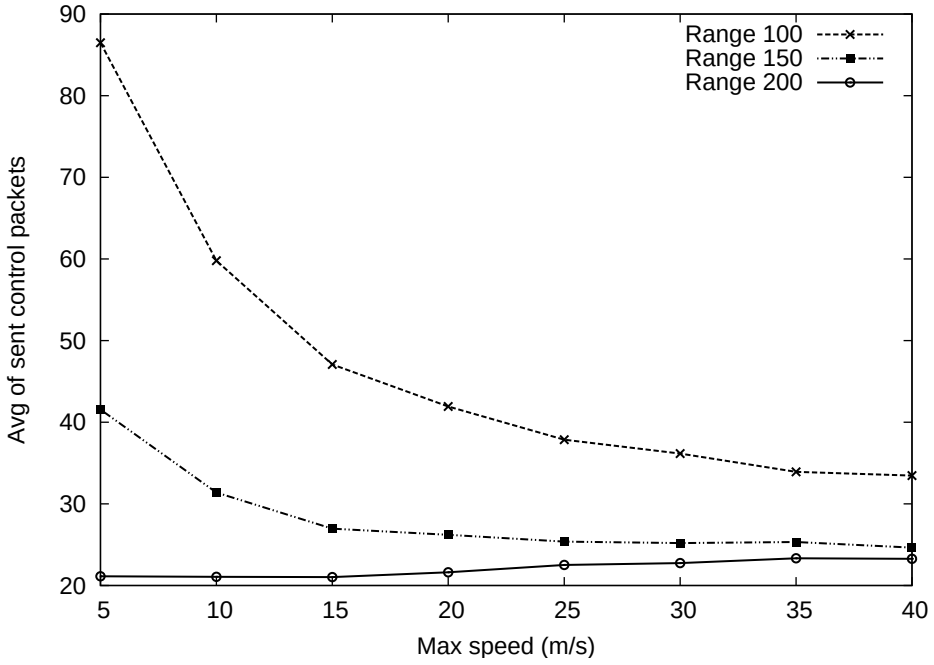

Figure 5.13: The average of sent control packets per node vs the max speed (m/s)

5.8 Summary

This chapter introduced $\diamondsuit RC$ and $\diamondsuit QC$ protocols in addition to the NM protocol which is used as an underlying service. Moreover, An intensive performance study for both $\diamondsuit RC$ and $\diamondsuit QC$ was carried out by simulation.

The structure of $\diamondsuit RC$ was explained in details by using two implementations. The first implementation was by using EGM as an added $\diamondsuit R$ service. The second one by not using any added $\diamondsuit R$ services. We carried out a detailed comparison between the performance of the two implementations. This comparison lead us to the following observations:

1. The first implementation incurs more delay than the second one especially in low densities.

2. The number of data transmissions in the first implementation is higher, but this number in the same implementation is considered to be reasonable; each node had 1.3 transmissions on average in the worst case scenario.

3. The first implementation had less number of transmitted control packets in all densities. The difference in the number of transmitted control packets cannot be tolerated.

The last observations lead to the result that $\diamondsuit RC$ using *EGM* gives a better performance than the $\diamondsuit RC$ which does not use any $\diamondsuit R$ services. Therefore, the former will be used in all future implementations.

$\diamondsuit QC$ was implemented also using *EGM*. It was shown that $\diamondsuit QC$ has a similar structure like $\diamondsuit RC$. The simulations prove that:

1. $\diamondsuit QC$ has the same cost like $\diamondsuit RC$ in terms of transmitted data packets.

2. Few extra control packet transmissions are incurred by $\diamondsuit QC$ comparing with $\diamondsuit RC$. The number of extra packets is very small in all densities.

3. $\diamondsuit QC$ takes some extra delay, this delay is reduced when densities are increased.

So the extra cost for $\diamondsuit QC$, comparing with $\diamondsuit RC$, is mainly in the delay where members need some extra time to be quiescent after the realization of a message m. This extra delay is reduced when density is increased.

The *NM* protocol was presented in details. This protocol is used to build and maintain the H-hop ($H \geq 1$) neighbourhood list and routes to group neighbours (when $H > 1$). This list and routes are used by $\diamondsuit RC$ and $\diamondsuit QC$ protocols.

NM builds routes on demand depending on a mesh routes structure. So routes are built just when they are needed to reduce the overhead. Moreover, *NM* can maintain redundant routes, which makes it robust against node mobility (as was shown in Figures 5.3 and 5.4).

Chapter 6

Consensus Protocol

The consensus protocol will be derived from that by Ezhilchelvan, Mostefaoui and Raynal [EMR01]. The latter will be referred to as the *EMR* protocol and works using dissemination primitives designed for a (connected) wired network. The new derived protocol will depend on the $\diamondsuit RC$ and $\diamondsuit QC$ for disseminating messages.

As was explained, the $\diamondsuit QC$ protocol will be used only by deciding members to disseminate decisions. So members will use the $\diamondsuit RC$ protocol for disseminating their estimates until they decide. The challenges which arise due to using the $\diamondsuit RC$ protocol will be handled here. To handle these challenges, the *EMR* protocol will have to be appropriately adapted.

So this chapter will briefly prsenet the *EMR* protocol, then the protocol derivation and challenges will be explained in details. Finally, a detailed performance study will be carried out to illustrate the cost of the derived protocol.

6.1 The *EMR* Protocol

The *EMR* protocol uses two communication facilities which operate within G and both are $\diamondsuit R$. These two facilities are:

1. A uniform reliable broadcast service which is denoated as *RBcast*. This service ensures that even if an operative member receives m and subsequently crashes, m is received by every correct member. It thus offers delivery guarantees stronger than $\diamondsuit QC$ multicast whose coverage guarantee applies only in executions where at least one correct member receives m.

2. A simple non-crash-tolerant broadcast service which denoted as *Bcast*. This

service ensures that all correct members receive m, if the broadcasting of m by the source member is not interrupted by a crash; otherwise, some correct members may receive m, while some others may not. This guarantee is stronger than that of $\diamondsuit RC$ in which even a correct member's m is not necessarily received by all other correct members.

The *EMR* protocol mostly uses *Bcast* and reserves *RBcast* only for the following two activities:

1. Each member N_i *RBcast*s its initial value to every other member at the start of the protocol and collects the values received through *RBcast* in a bag called *Value-Bag* or $VBag_i$.

2. If N_i decides, it *RBcast*s its decision to others so that the latter can decide upon receiving the decision.

The main part of the execution involves n-to-n message exchanges until some member can decide. It is briefly described as follows. It proceeds in rounds, with each round having two phases. During the first phase of a round, every member broadcasts a value called its *estimate*, *est* for short, and waits to obtain estimates from a majority of members in G. If estimates from a majority are the same, that identical value is adopted as the member's new estimate; otherwise a default value $\perp$ (different from any possible estimate) is taken as the new estimate.

During the second phase, as in the first, members exchange their estimates and wait to obtain estimates from a majority of members of G. If a member obtains the same value from a majority, it decides on that value and *RBcast*s its decision. Otherwise, it proceeds to the first phase of the next round with a new estimate computed as follows: if it has obtained at least one v, $v \neq \perp$, v is its new estimate; else, i.e., if all the values obtained are $\perp$, the member chooses its new estimate randomly from its $VBag$.

The *EMR* protocol achieves termination with probability 1 by using two facts: all operative members eventually have identical $VBag$ (thanks to the uniform delivery property of *RBcasts*) and there is a small probability that all members randomly select the same value from their $VBag$s and start a round with the same estimate. The correctness proofs of [EMR01] show that the protocol must eventually terminate.

6.1.1 Protocol Derivation and Challenges

The new protocol is derived from the EMR protocol by addressing the implications of replacing $RBcast$ and $Bcast$ with $\diamondsuit QC$ and $\diamondsuit RC$ multicasting respectively; moreover, the use of $\diamondsuit QC$ multicasting is kept to a minimum.

6.1.1.1 Using $\diamondsuit QC$ Multicast

It is done only for disseminating the decision value. By the definition of consensus, the value decided must be the same, irrespective of which or how many members decide. Therefore, several deciding members initiating $\diamondsuit QC$ multicast need not be distinguished based on sender identifiers and can be easily optimized to equivalent of handling a single $\diamondsuit QC$ multicast.

The new consensus protocol does not require members to exclusively disseminate their initial estimates for the sake of building identical $VBag$. However, it requires, just like EMR, that operative members eventually have identical $VBag$. While EMR achieves this requirement through accumulation of values disseminated by $RBcast$, the new protocol does it through elimination of values from $VBag$ so that all operative members eventually have identical $VBag$ containing a single value. The scheme is explained below.

Members make a random selection at the end of each phase, concurrently to their attempts to decide based on est exchanged as per the logic of the EMR protocol. Each N_i maintains a variable $cand_i$ which is its input candidate for the random selection process. $cand_i = est_i$ for phase 1, round 1.

At the start of each phase, N_i sends $cand_i$ together with its est_i. Once N_i receives $\{est, cand\}$ pair from a majority of members (including its own), $VBag_i$ becomes the set of all distinct $cand$ values N_i received in the *current* phase. A value selected randomly from $VBag_i$ becomes the new $cand_i$ which will be N_i's input candidate for the selection to be held at the end of next phase. It is later shown that if random selections are repeated frequently enough, $VBag$s of operative members decrease in size and eventually become identical singleton sets, returning the same $cand$ after a 'random' selection.

6.1.1.2 Using $\diamondsuit RC$ Multicast

At least $(n - f)$ members receive a $\diamondsuit RC$ multicast m. In the worst case, f of those that $\diamondsuit RC\_received$ m may crash, leaving only $(n-2f)$ correct members with m. This worst-case possibility may block a correct member receiving m from a majority of members, when $n - 2f \leq \frac{n}{2}$, i.e., for all n, $2f + 1 \leq n \leq 4f$. Note that if a member cannot $\diamondsuit RC\_receive$ $\{est, cand\}$ pair from a majority of members, it cannot complete a phase. (Completing a phase means deciding or moving onto the next phase/round). A deadlock arises when no member can complete a phase and this is illustrated by the scenario described below.

Consider $G = \{N_1, N_2, N_3\}$ and $f = 1$. Say, N_1 and N_2 $\diamondsuit RC\_mcast$ a message for round r and phase ph, denoted as (r, ph) message for short. Let (r, ph) messages from N_1 and N_2 be $\diamondsuit RC\_received$ by $\{N_1, N_3\}$ and $\{N_2, N_3\}$ respectively. Suppose that N_3 crashes after $\diamondsuit RC\_receiving$ these messages but before processing them and therefore before $\diamondsuit RC\_mcasting$ any (r, ph) message of its own. Both N_1 and N_2 cannot now receive two (r, ph) messages, if their own (r, ph) messages have been discarded by the $\diamondsuit RC$ multicast protocol. $\diamondsuit RC$ discards a message when it is known that this message was received by at least $n - f$ members. In this scenario, it is enough for a message to be received by 2 members to become realized. So if N_1/N_2 knows that N_3 has received its (r, ph), it realizes its (r, ph). Therefore, both N_1 and N_2 will cease the multicast of (r, ph), although each of them has only its own $\{est, cand\}$.

To break the deadlock, it is necessary for operative members to repeat their $\diamondsuit RC\_mcasting$ of (r, ph) messages at regular intervals, if they judge themselves being unable to receive (r, ph) messages from a majority of members. In the above scenario, the (r, ph) messages $\diamondsuit RC\_mcast$ by N_1 and N_2 after N_3 has crashed will allow N_1 and N_2 to make progress.

This repetitive $\diamondsuit RC_mcast$ing of (r, ph) messages must be done in a judicial manner, mindful of overheads involved. Our protocol manages this using a time-driven mechanism that can also be event-driven one when $2f + 1 < n \leq 4f$: a member distinquishes its $\diamondsuit RC$ multicasts for a given (r, ph) message using a message field, called the *attempt sequence number* and denoted as α, $\alpha \geq 1$. The $\diamondsuit RC$ multicast system will treat $\diamondsuit RC$ multicasts from a given member with distinct α as distinct mulitcast messages. Every time a member $\diamondsuit RC\_receives$ a (r, ph) message,

```
Function consensus (v_i)
 {
(1) r_i = ph_i = α_i = 1; est_i = cand_i = v_i;
(2) activate Thread C(r_i, ph_i, α_i, est_i, cand_i);
(3) wait until ◇QC_receive (decision, v) occurs;
(4) return (v);
 }
```

Figure 6.1: The Consensus Function

it increments a *count* by 1. When the *count* reaches $(n - 2f)$, the (r, ph) message will be $\diamondsuit RC\_mcast$ after Δ time and the *count* is decremented by $(n - 2f)$.

In the scenario considered earlier with $G = \{N_1, N_2, N_3\}$ and $f = 1$, only Δ-driven mechanism is effective as $n = 2f + 1$: both N_1 and N_2 will perform a $\diamondsuit RC\_mcast$ once every Δ interval until N_3 crashes or no longer receives the $\diamondsuit RC\_mcasts$ from N_1 and N_2.

It is proved in our work in [AE10] that if a correct member begins a phase ph of round r, then some correct member $\diamondsuit RC\_receives$ (r, ph) from a majority of members, so long as no correct member is decided. The proofs make no assumption as to how long a faulty member, such as N_3 in the earlier scenario, can take to crash or to stop receiving correct members' repeated $\diamondsuit RC$ multicasts; they only require the interval to be finite.

6.1.2 The Protocol

The protocol is expressed in Figure 6.1 as a function that takes a member N_i's proposed estimate and returns the decision v (line 4). It assumes the use of $\diamondsuit QC$ and $\diamondsuit RC$ protocols which respectively provide $\diamondsuit QC\_mcast(m)$ and $\diamondsuit RC\_mcast(m)$ for multicasting m, and $\diamondsuit QC\_receive()$ and $\diamondsuit RC\_receive()$ for receiving a multicast m exactly once.

The round-based activities are carried out by thread $C()$ which is activated in line 2 of Figure 6.1 for phase $ph = 1$ of round $r = 1$, $(r = 1, ph = 1)$ for short. Thread activation supplies five input parameters which are all initialized in line 1 of Figure 6.1, where, it should be noted, $cand_i$ and est_i are both set to N_i's own proposal v_i and α_i is set to 1.

Thread $C()$ is designed to die after either deciding or spawning a new instance of itself for a later phase. If decision is made, it is $\diamondsuit QC$ multicast which, when $\diamondsuit QC\_received$ (line 3), terminates Function consensus.

```
 Thread C(r_i, ph_i, α_i, est_i, cand_i)
 {
(1)  while (true) do
     {
(2)    m = (r_i, ph_i, α_i, est_i, cand_i); ◇RC_mcast (m);
(3)    Bag_i = {}; VBag_i = {}; count_i = 0;
(4)    while (|Bag_i| ≤ n/2) do
       {
(5)      while (count_i < n − 2f) do
         {
(6)        wait until (r_j, ph_j, α_j, est_j, cand_j)
             ◇RC_received: (r_j = r_i ∧ ph_j ≥ ph_i) ∨ (r_j > r_i);
(7)        if ((r_j = r_i ∧ ph_j > ph_i) ∨ (r_j > r_i)) then
           {
(8)          activate Thread C(r_j, ph_j, 1, est_j, cand_j); die;
           }
(9)        count_i + +;
(10)       if (cand_j ∉ VBag_i) then enter cand_j in VBag_i;
(11)       if (<j, est_j> ∉ Bag_i) then enter <j, est_j> in Bag_i;
(12)       if (|Bag_i| > n/2) then { exit; }
         }
(13)     if (count_i ≥ n − 2f ∧ |Bag_i| ≤ n/2) then
         {
(14)       count_i = count_i − (n − 2f); α_i = α_i + 1;
(15)       schedule ◇RC_mcast (r_i, ph_i, α_i, est_i, cand_i)
               at clock + Δ;
         }
       }
(16)   cancel any pending ◇RC_mcast();
(17)   cand_i = PickRandom(VBag_i);
(18)   if (ph_i = 1) then
       {
(19)     if (v = est_j for all <j, est_j> in Bag_i) then est_i = v;
(20)     else est_i = ⊥;
(21)     ph_i = 2; α_i = 1;
       }
(22)   else if (ph_i = 2) then
       {
(23)     if (v = est_j ∧ v ≠ ⊥ for all <j, est_j> in Bag_i) then
         {
(24)       ◇QC_mcast (decision, v); die;
         }
(25)     else if (v = est_j ∧ v ≠ ⊥ for some <j, est_j> in Bag_i) then
(26)       est_i = v;
(27)     else est_i = cand_i;
(28)     r_i = r_i + 1; ph_i = 1; α_i = 1;
       }
     }
 } // end Thread C()
```

Figure 6.2: Pseudo-Code for Consensus Thread

The pseudo-code for thread $C()$ of some $N_i \in G$ for a given $(r,\ ph)$, $r \geq 1$ and $ph \in \{1, 2\}$, is presented in Figure 6.2. It can be understood in three parts: N_i $\diamondsuit RC$ multicasting own $\{est, cand\}$ (line 2), awaiting $\{est, cand\}$ pairs to be $\diamondsuit RC\_received$ from a majority of nodes (lines 3-15) and acting on the $\{est, cand\}$

pairs received (lines 16-28).

To deduce the termination of waiting in part 2, Bag_i is maintained for storing est_j that was $\diamondsuit RC\_received$ from $N_j \in G$, as a 2-tuple $<j, est_j>$. Having initialized Bag_i, $VBag_i$ and $count_i$ (in line 3), the thread waits (line 6) to $\diamondsuit RC\_receive\ m_j$ from any $N_j \in G$ for some (r_j, ph_j) that is either future to or the same as (r_i, ph_i).

A future m_j will have either ($r_j = r_i$ and $ph_j > ph_i$) or $r_j > r_i$ (line 7) and will expedite the execution: a new thread instance is created with future m_j as the input (except for α_j which is initialized to 1), and the current thread dies (line 8).

If m_j is current with $r_j = r_i$ and $ph_j = ph_i$, $count_i$ is incremented (line 9); $VBag_i$ and Bag_i are updated, if $cand_j$ and $<j, est_j>$ are not already present, respectively (lines 10-11). If $count_i \geq n - 2f$ without termination of part 2 (lines 12-13), then a repeat of $\diamondsuit RC\_mcast(m)$ after Δ time is scheduled (line 15), once $m.\alpha$ is increased by 1 and $count_i$ decreased by $(n - 2f)$ (line 14).

Once Bag_i has more than $\frac{n}{2}$ entries, part 3 begins: any pending $\diamondsuit RC\_mcast(m)$ is canceled (line 16) and $cand_i$ is set to a random value picked from $VBag_i$ (line 17). The rest of part 3 faithfully implements the EMR logic as described in §6.1 for both $ph = 1$ and 2, except for one aspect: when $ph = 2$, if est_i has to be a random selection then est_i is simply set to $cand_i$ (line 27). The rest of part 3 (lines 18-28) is shown below in more details.

When $ph = 1$: (i) If all received estimates from a majority are the same and equal to v, v is adopted as the member's new estimate (line 19); otherwise $\perp$ is taken as the new estimate (line 20), and (ii) ph_i is set to 2 and α_i to 1 (line 21).

When $ph = 2$: If all obtained estimates from a majority are the same and equal to v ($v \neq \perp$) (line 23), N_i decides on v and $\diamondsuit QC_mcast$s the dcision, then the current thread dies (line 24). If, otherwise, N_i has received at least one v, $v \neq \perp$ (line 25), it takes v as its new estimate (line 26). Otherwise, $cand_i$ is taken as the new estimate (line 27). Finally, r_i is incremented by 1, and both ph_i and α_i are set to 1 (line 28).

6.2 Performance Study

The simulation parameters used in this section are the same parameters which were used in the last chapter. A total of 50 nodes ($\mathcal{S} = 50$) were randomly placed in a fixed size terrain of 1000m x 1000m. At the start of each run, 10 nodes are chosen

randomly to be group members ($G = 10$). The group members do not leave the group during the run.

In the same way as in the last chapter, each run of simulation was repeated 1000 times using distinct random seeds. Moreover, every run started after 1000 seconds of node movement to avoid any initial bias in node placement.

The value of β in $\diamondsuit RC$ and $\diamondsuit QC$ protocols was varied as: 2 and 5 seconds. No significant difference in performance was observed. So we present our results using $\beta = 5$, then some chosen graphs which use $\beta = 2$ are added to demonstrate the similarity in performance. So all presented graphs use $\beta = 5$ unless it is explicitly stated.

Consensus protocol always starts with each node proposing a distinct initial estimate. Also, at the start, three nodes are randomly chosen for crashing at randomly chosen moments. Δ in the consensus protocol was chosen to be 10 seconds - a large value because the failure scenario (see § 6.1.1.2) that calls for repeated $\diamondsuit RC$ multicasting of a given (r, ph) message was judged to occur rarely. (It was never observed in our study).

The used value, in $\diamondsuit RC$ and $\diamondsuit QC$, for H in $Neigh_H$ (ii) is 2 when a node has an unrealized multicast, or (ii) is set to 1, once all on-going multicasts are realized. For $H > 1$, members update $Neigh_H$ every 2 seconds interval by transmitting *Ghello* packet. The number of allowed *Ghello* losses is 2; a neighbour is deleted when its *Ghello* is not received within $3 \times 2 = 6$ seconds. Finally, when a node begins to execute a given phase, ph of round r, it kills any on-going $\diamondsuit RC$ multicasts for an earlier phase.

$\diamondsuit RC$ and $\diamondsuit QC$ use *EGM* as their $\diamondsuit R$ service. *EGM* uses the value 6 for τ ($\tau = 6$) and the values for ϕ are the same as the ones listed in Table 4.2.

We measured 5 performance metrics for 3 different densities, and 8 different maximum node speeds. They are presented in three categories: number of rounds, time overhead and packet overhead.

1. Average number of rounds: The number of rounds, measured as an average over 1000 runs, taken to reach consensus.

2. Time overhead: It was measured in terms of (i) time taken for the first members to decide, and (ii) time taken for members to become *totally quiescent*. The latter occurs when all operative members have identical RV for a multi-

cast m and hence not even a control packet (such as K_pkt or $Ghello$) needs to be unicast for that m.

3. Packet overhead: The average of control and data packets per node which is measured as the total number of control/data packets transmitted by all nodes until total quiescence, divided by $|\mathcal{S}| = 50$. A data packet refers to any m containing (est, $cand$) pair or the decision.

We also measure the packet overhead until the first members to decide. This overhead has the same definition used for packet overhead in 3, but here the transmitted control/data packets are calculated until the first members decide.

Figure 6.3 shows that members take less number of rounds to reach consensus when density is lower. This section will prove that the number of rounds is the only metric which benefits from low density. The node speeds have no important effect on the number of rounds.

It appears that when the network is denser, more members receive each other's estimates and *simultaneously* complete a round/phase. As the density drops, few members complete a round/phase much ahead of the rest, resulting in them expediting the slow ones and enforcing their random choice on to the slow ones. Consequently, estimates of different members converge faster and consensus is reached in fewer rounds.

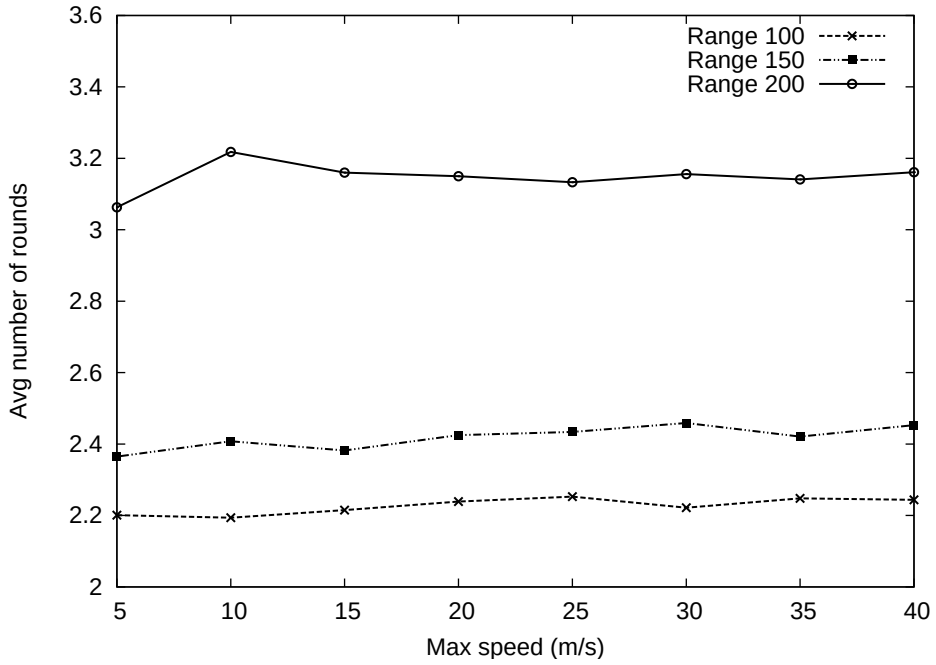

Figure 6.3: Average number of rounds vs the max speed (m/s)

Figures 6.4 and 6.5 show the time for first members to decide and time for the total quiescence repectively. These graphs have the same trends for the node density and the node speed. They show that the lower the density, the longer the latencies. Despite the fewer number of rounds in lower densities, these densities incur longer delay. Note that the latencies which shown in Figure 6.4 and 6.5 are affected by latencies in $\diamondsuit RC$ and $\diamondsuit QC$ protocols repectively because these protocols are used here. So in the same way like in $\diamondsuit RC$ and $\diamondsuit QC$, increasing the node speed (up to a threshold) helps reduce the latencies. This is because increasing the speed casuses the $(B, H \geq 2)$-*Connectivity* to be formed quickly; beyond the threshold, connectivity does not last for the needed B duration due to quick node movement. Longest latencies are observed when density and maximum speed are at their smallest, 1.6 and 5 m/s: it takes 20 and 34 minutes for the first node to decide and for total quiescence, respectively. Doubling the density value reduces latencies by more than half at all node speeds. So a dramatic decrease in latencies is observed when increasing densities because in higher densities connectivity between members is formed quickly.

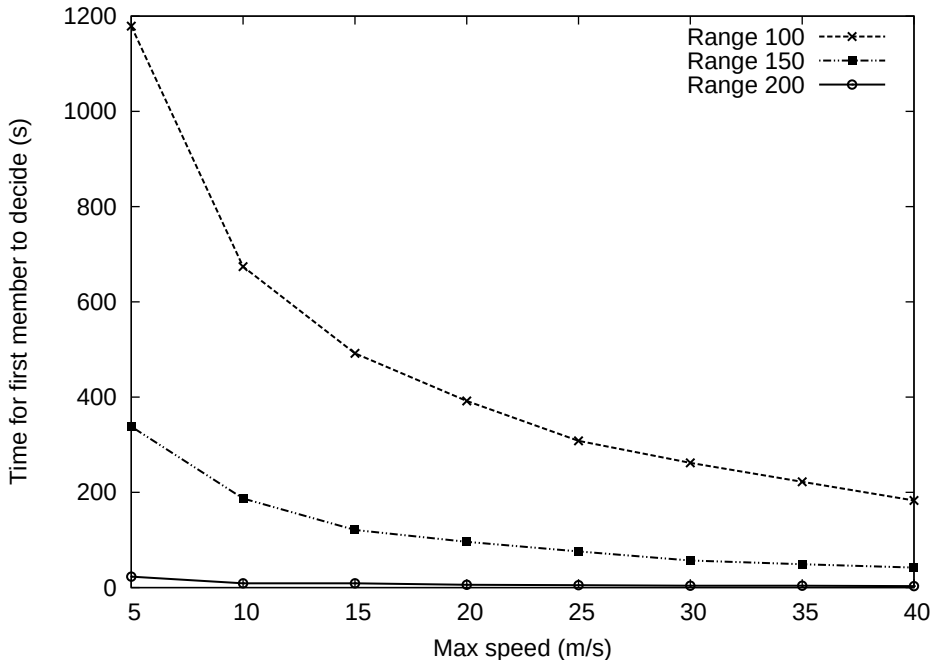

Figure 6.4: Time until the first members decide vs the max speed (m/s)

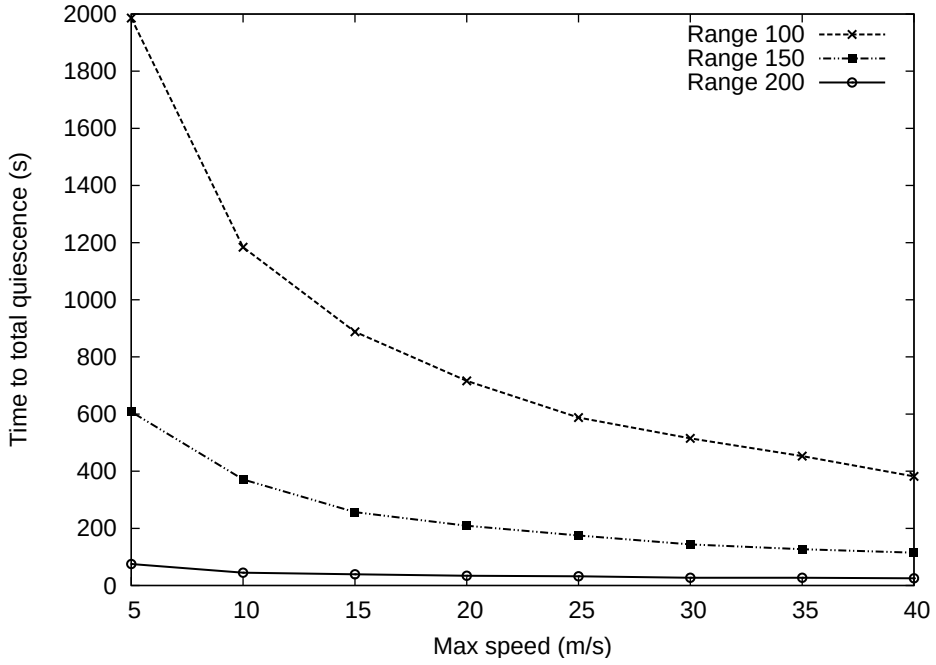

Figure 6.5: Time until total quiescence vs the max speed (m/s)

The average of data and control packets, transmitted per node until total quiescence, are presented in Figures 6.6 and 6.7 respectively.

It is shown that data packet overhead is fairly constant over node speeds. This is because the node speed has no important effect on the data packet overhead of $\diamondsuit RC$ and $\diamondsuit QC$ which are used here. The node density has a significant influence on the data packet overhead. Despite the fact that higher densities require more number of rounds, they incur less data packet overhead than lower densities. This is because at higher densities, (B, H)-*Connectivity* is formed quickly. So members can receive each other's estimates, and so move to advanced phase, more quickly. As was mentioned before, when a member begins to execute a given phase, ph of round r, it kills any on-going $\diamondsuit RC$ multicasts for an earlier phase. This causes the number of killed $\diamondsuit RC$ multicasts to be greater at higher densities. So the data packet overhead will be smaller.

Control packet overhead, in Figure 6.7, shows a similar trend as the latencies; the longer it takes to reach consensus, the more control packets are being expended. This is because $\diamondsuit RC$ is used here and the longer it takes to reach consensus, the longer $\diamondsuit RC$ needs to be working; As was shown in the last chapter, longer $\diamondsuit RC$ latencies incur more control packet transmissions. In fact the control packet overhead for $\diamondsuit RC$ has the same trend like Figure 6.7.

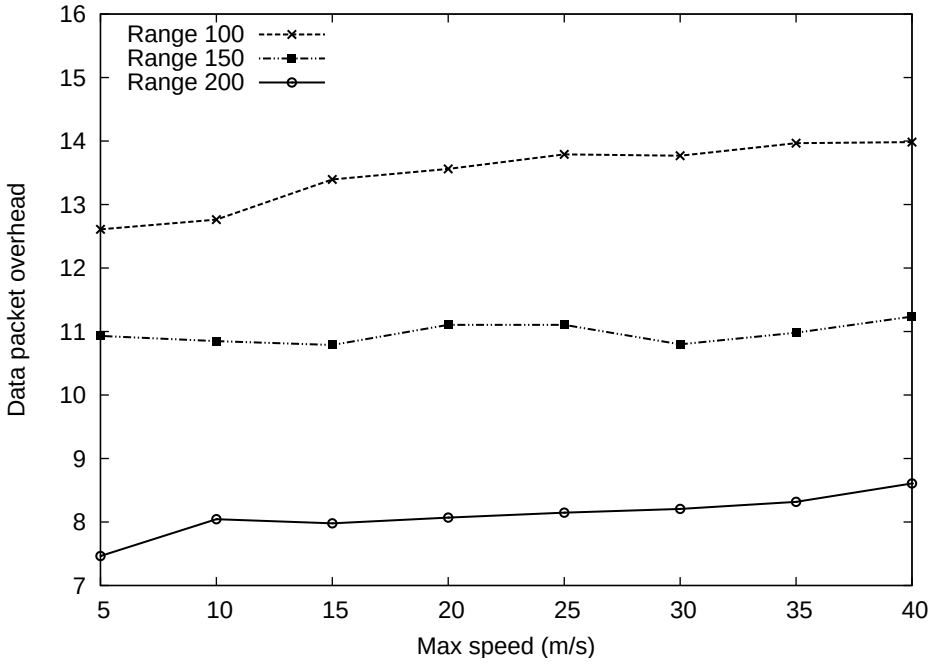

Figure 6.6: The average of transmitted data packets per node unitil total quiescence vs the max speed (m/s)

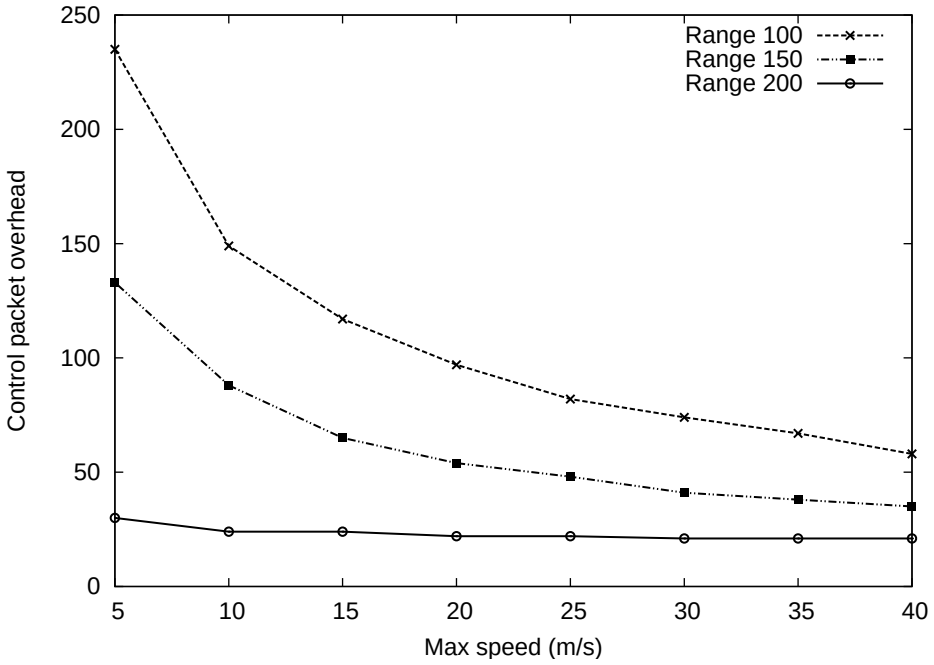

Figure 6.7: The average of transmitted control packets per node unitil total quiescence vs the max speed(m/s)

Figures 6.8 and 6.9 show data and control packet overhead until the first members to decide. These Figures show the same trends like 6.6 and 6.7 respectively. Figures 6.6 and 6.7 have some extra overhead which is required for members to reach consensus after the first members decide (as expected).

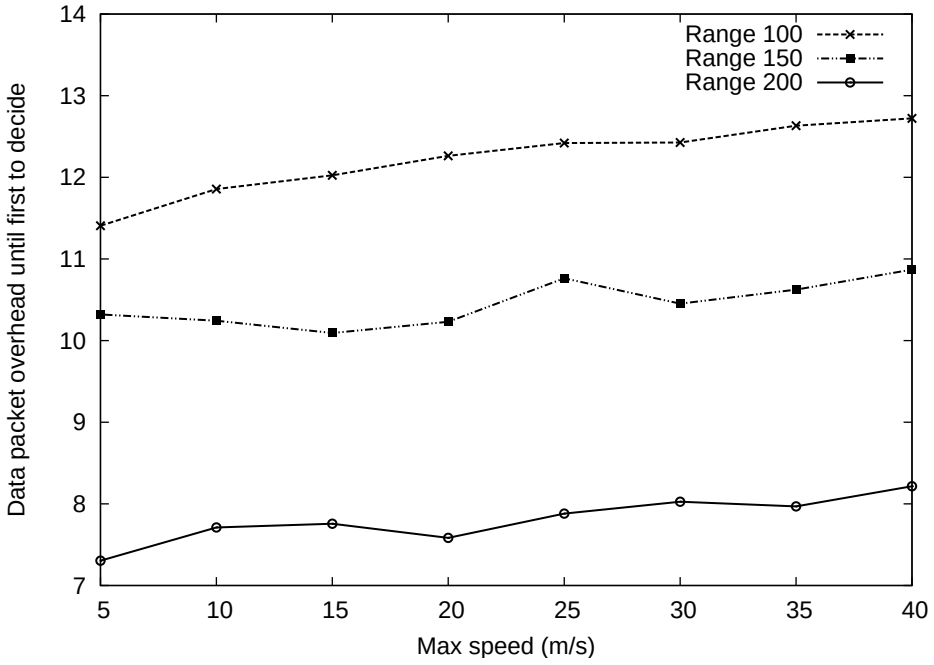

Figure 6.8: The average of transmitted data packets per node until the first members decide vs the max speed (m/s)

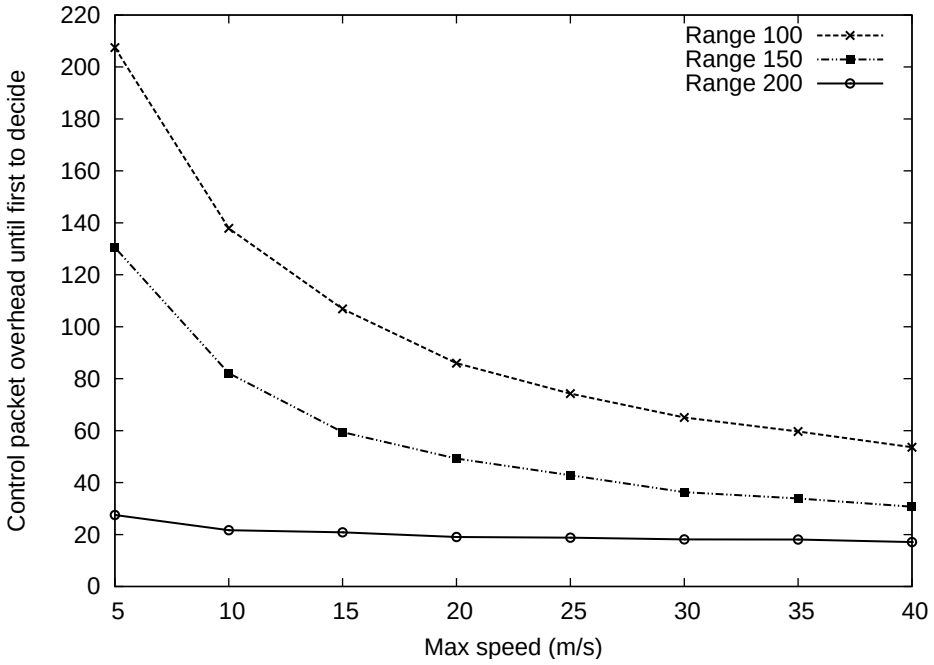

Figure 6.9: The average of transmitted control packets per node unitil the first members decide vs the max speed(m/s)

6.2.1 The Performance Using $\beta = 2$

We show in this section some of our results using $\beta = 2$. Since the purpose is to discuss the influence of different values of β, we choose to show the metrics which

are more connected to the time taken to decide/reach consensus. So we will show here the following metrics:

1. Time taken for the first members to decide.

2. Time taken for members to become totally quiescent.

3. The average of transmitted control packets until total quiescence.

Comparing Figures 6.10 and 6.11 with Figures 6.4 and 6.5, respectively, shows that there is no important difference in the time overhead using $\beta = 2$ and $\beta = 5$. Moreover, comparing Figure 6.12 with Figure 6.7 shows that there is no significant difference in the control packet overhead between using the two different values of β.

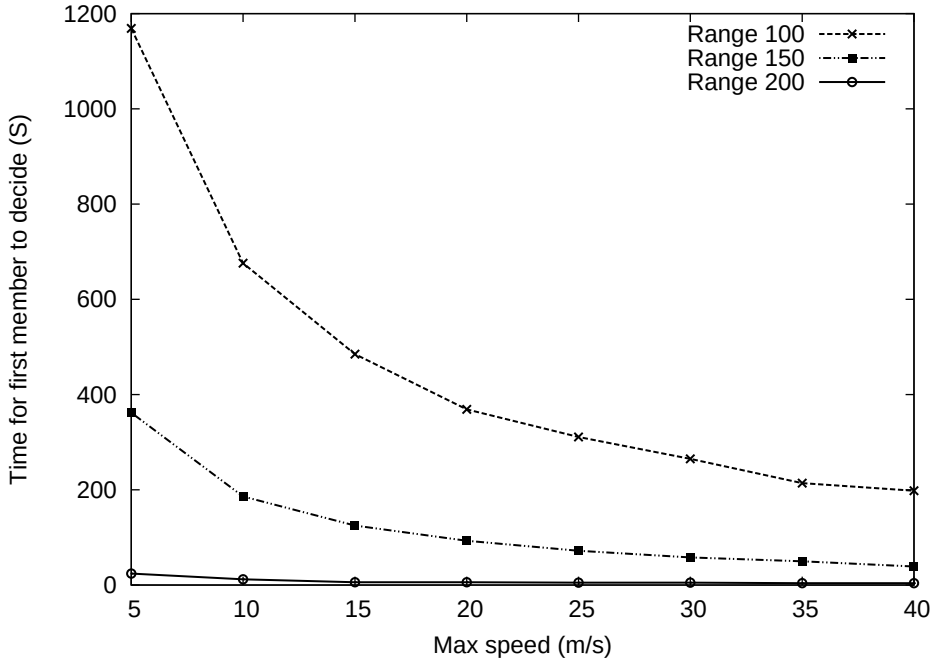

Figure 6.10: Time until the first members decide vs the max speed (m/s)

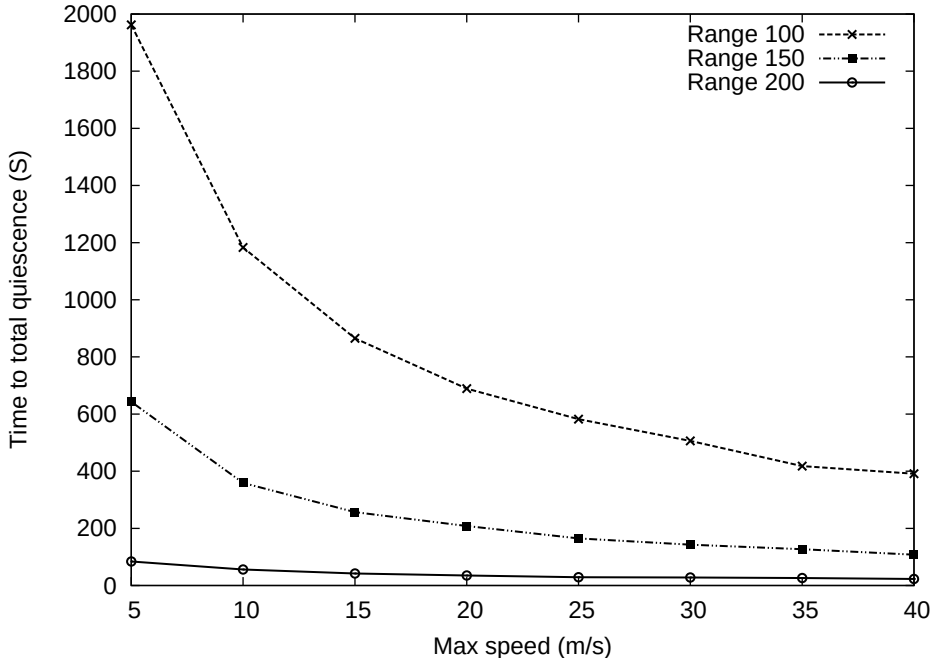

Figure 6.11: Time until total quiescence vs the max speed (m/s)

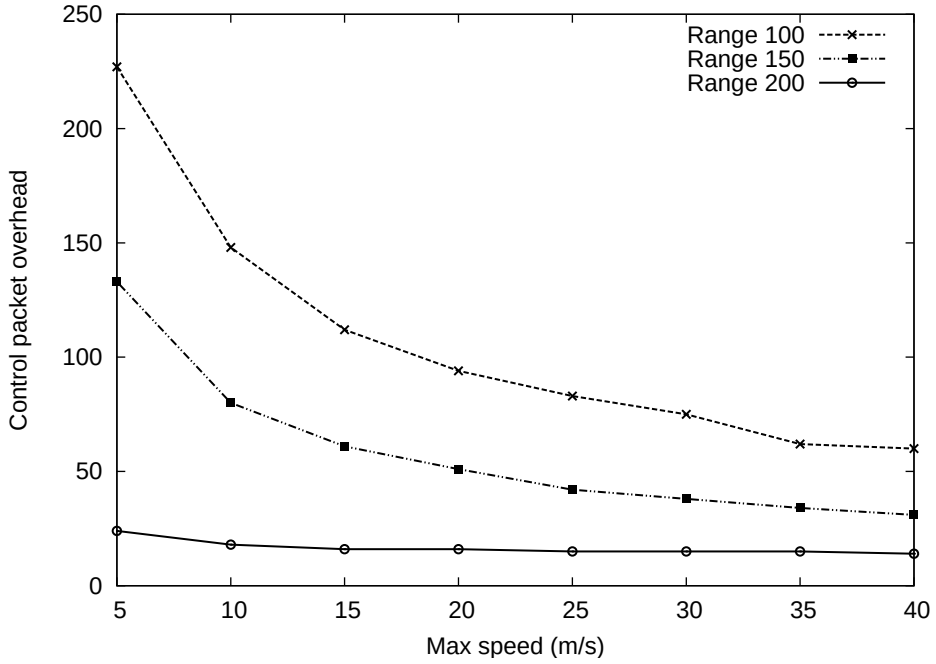

Figure 6.12: The average of transmitted control packets per node unitil total quiescence vs the max speed(m/s)

6.3 Compariosn Study

In this section, we compare the performance of our consensus protocol, as an example of randomized consensus protocol, against one of the leader-based consensus protocols which were introduced in Chapter 2. So next, we introduce the leader-based

approach in sparse MANETs and identify the design challenges that arise.

6.3.1 Features of Leader-based Consensus Protocols

Our randomized consensus protocol proceeds in rounds, each round is made up of two phases. In the same way, a leader-based consensus protocol proceeds in rounds each of which is made up of two phases. In the leader-bases protocols, however, consultation messages are sent to the leader which, in turn, multicasts its reply to all group members.

In the best case for a leader-based protocol, a decision can be reached at the end of round 1 if an operative leader is:

(a) able to communicate with enough number of members, and

(b) trusted by these members long enough while it carries out the leadership tasks.

The latter feature is one of the advantages of a leader-based protocol over a randomized one. A leader-based protocol has one more advantage: if the leader of a round crashes or, in the case of sparse MANETs, remains unable to receive enough consultation messages from other members, the protocol has a built-in procedure to switch to another leader. So if a leader cannot make progress (e.g. cannot achieve (a) above) during any round, it can be replaced by another leader.

So a leader-based protocol can circulate the leadership between a group of members. This allows this protocol to tackle any deadlock that might be caused by crashes or poor leader connectivity. However, the down side to the latter advantage is the difficulty in choosing the suitable timeout before changing a leader, this timeout will be referred to as *suspicion timeout*. The chosen duration for the suspicion timeout has a big effect on the protocol behaviour; choosing an unduly small timeout duration subverts (b) above, even if the current leader is being assisted by the MANET to achieve (a).

So, we experimentally explore the following question: how helpful is a sparse MANET in letting a leader accomplish (a) while (b) is never undermined? (Thus, in our experiments, a leader change occurs *only* if the MANET has not helped the current leader to achieve (a)). Experiments confirm that the MANET is least helpful; This, we argue, requires the messages be sent using opportunistic forwarding protocols, such as $\diamondsuit QC$ or $\diamondsuit RC$ protocols we have employed for our consensus protocol.

The necessity to use opportunistic forwarding protocols have two implications. First, consensus latencies would be longer even if (b) above is never undermined. Secondly, choosing a timeout duration that never undermines (b) above is hard. Expanding on the second implication, we conclude that a rigorous performance comparison between a leader-based protocol and ours cannot be done in any meaningful manner. We end with providing a qualitative comparison on achievable consensus latencies, subject to a few caveats.

6.3.1.1 Leader-based Performance Study

We chose to simulate Fast Paxos [Lam06] which is a leader-based protocol (Chapter 2 has some details about Fast Paxos). The target is to assess how swiftly this sparse MANET supports the protocol by being well-connected for a sufficiently long time in order that some leader manages to complete just one round. In other words, we assess the efficiency of Fast Paxos in the absence of any support for opportunistic forwarding.

So we simulated this protocol in a MANET of density 1.6 for speed 5 m/s. In our experiment, the leadership role is circulated amongst operative members after it is known that no message is in transit and the current leader still remains unable to discharge its responsibilities.

All multicast communications were performed using flooding whose capacity for delivering messages is as effective as the connectivity that exists during the brief period that a flood lasts. The suspicion timeout was chosen to be 10 seconds, to eliminate any influence of it being too small. in the same way like other simulation experiments, each Paxos simulation was carried out 1000 times; each run was terminated either as soon as a member decides or 200 leader changes have occurred with no member deciding. Figure 6.13 shows the cumulative distribution function (CDF) of the number of leader changes before reaching a decision.

Referring to Figure 6.13, we notice that 200 leader changes are enough to reach decision only in 69% of the runs. Furthermore, only in a few runs the decision was reached after a reasonably small number of leader changes: for example, the probability that a decision is reached within 20 leader changes is 0.15 ($P(\leq 20) = 0.15$). The experimental data also suggests (not obvious from Fig 6.13) that the decision was reached with the first, the second and the third leader in 2%, 1% and

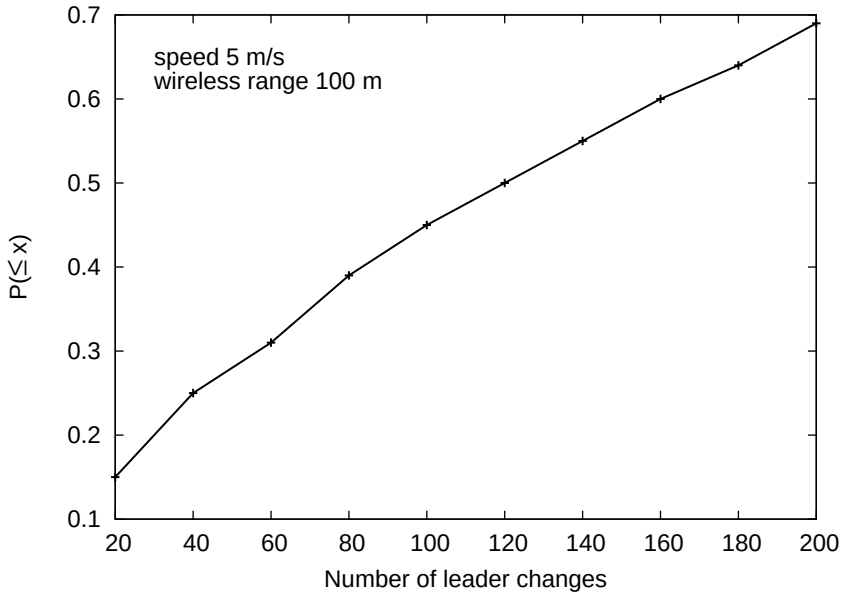

Figure 6.13: CDF of number of leader changes

1% of 1000 runs. That is, a sparse MANET of 1.6 density is only 4% reliable in providing the necessary connectivity so that the decision can be reached in at most 2 leader changes.

Another important observation can be derived from what did not happen in our experiments: in 31% of the 1000 runs, no decision was reached even after 200 leader changes. Note that the nature of the leader-based protocols is such that when the leadership is changed, the new leader cannot build on the incomplete work done by the past leader(s), but must start its work from scratch. (See [Lam06] for more details). So a leader-based protocol is memoryless when it comes to leader changes: the probability of the i^{th} leader deciding does not depend on i, but on how it is favored by the network connectivity patterns when it is discharging its leader responsibilities.

Of course, if the MANET were to eventually bestow its favor to some leader at some time, then $P(\leq \lambda) = 1$ for some large number λ of leader changes. On the other hand, the MANET may not be eventually obliging and as a result $P(\leq \lambda) < 1$ as $\lambda \to \infty$. In that case, $1 - P(\leq \lambda)$, $\lambda \to \infty$ refers to the tail probability that a decision is never reached. Based on the experiments that limit λ to 200, we can conclude that the opportunistic forwarding protocols are also an absolute necessity for leader-based consensus protocols when the MANET is sparse.

6.3.1.2 Implementation Support and Suspicion Timeout

We identify four requirements for implementing a leader based protocol:

- A $\diamondsuit RC$ multicast protocol for the leader to disseminate its phase-1 or phase-2 m. Though blocking due to post-delivery crashes can be a problem, this does not necessarily require repeating of $\diamondsuit RC$ multicasts. However, the (single) leader is to make several attempts in a given phase, it would increase the coverage as the MANET topology is likely to change in between attempts.

- A unicast protocol that employs opportunistic forwarding so that a non-leader could respond to the leader's phase-1 or phase-2 m with a small overhead. We refer the reader to [Zha06] for choosing an effective unicast protocol.

- A $\diamondsuit QC$ multicast protocol for the leader to disseminate its decision which would occur if it receives responses to its phase-2 m from a majority.

- The duration of the suspicion timeout, STO for short, which a non-leader member, say, N_i sets after it has responded to the leader's phase-1 or phase-2 m; if phase-2 m or the decision m is not received before the timeout expiry, N_i will act as the leader if it is the next leader candidate; otherwise it waits on STO for the next leader to act. (All leader-based protocols pre-arrange the leadership sequence).

The duration of STO must be at least as large as the sum of an upper bound on the $\diamondsuit RC$ multicast protocol latencies and an upper bound on the unicast protocol latencies. Otherwise, non-leader members will falsely suspect a leader and would deny it the chance to complete the round which might have occurred if only more time had been given to it.

Establishing upper bounds on multicast and unicast latencies can be done through simulations or analytical estimations, but both require MANET characteristics to be fully known in advance and is specific to the multicast and unicast protocols used. More specifically, latencies depend on:

- The connectivity patterns that emerge while the multicast or unicast is in progress. i.e., the MANET characteristics, such as density, node speed etc.; and,

- The protocol mechanisms employed to exploit the emerging connectivity patterns for message propagation and the values chosen for protocol parameters such as β, H in $Neigh_H$ etc.

Even when all MANET and protocol parameter values are known, analytical estimation of latency bounds (in terms of these parameters) is a non-trivial task and we refer the reader to [Asp10] which is, to the best of our knowledge, the latest work to take on this problem.

In practice, however, MANET characteristics can change over time. For example, as devices run low on battery, they may reduce their transmission range causing a drop in network density. As our experiments indicate, a small drop in density can adversely alter the latency behavior. So, if MANET characteristics change past deployment, revising the bounds, be through simulations or estimations, is difficult since the global MANET state needs to be periodically assessed.

Thus, identifying an appropriate value for *STO* and adapting it when the MANET characteristics change, are a difficult (if not impossible) task. When the chosen *STO* is, or subsequently becomes, unduly small relative to the prevailing multicast and unicast latencies, false suspicions lead to unwarranted leader changes which in turn increase consensus latencies and message overhead. This also means that choosing the right *STO* is not only hard and but the performance of a leader-based protocol is also sensitive to the chosen *STO*.

6.3.1.3 Randomized vs. Leader-Based Protocols

Our consensus protocol is free of *STO* and Δ is the only timeout it uses. But these two timeouts do not impact performance identically. When *STO* is chosen to be large (relative to the prevailing latencies), the leader crash (a rare event) will be detected late; similarly when Δ is chosen to be large (again, relative to the prevailing latencies), it will take a long time to deal with blocking due to post-delivery crashes (also a low-probability event). Here, the similarities end. A smaller *STO*, as indicated earlier, will cost messages and prolong the time taken to reach consensus; on the other hand, a smaller Δ will cost only messages and speed up the release of any blocking due to post-delivery crashes.

Thus, the leader-based protocols and ours (or the randomized protocols in general) are quite distinct in several aspects:

- Design: centralized *versus* decentralized;
- Dissemination support: multicasts and unicasts *versus* multicasts only;
- Timeout Used: STO with or without Δ *versus* Δ only; and,
- Performance impact due to unduly small timeouts: increase of both latency and message overhead *versus* increase of message overhead only.

Therefore, a *rigorous and quantitative* performance comparison requires an agreed framework for making certain choices, listed here in the increasing order of difficulty: choosing (i) a leader-based protocol for implementation, (ii) an effective unicast protocol, and (iii) appropriate values for protocol parameters that are not necessarily present in both these consensus protocols. In the absence of such a benchmark, an attempt to compare their performance risks being seen to favor one approach over the other. Therefore, we below make an *informal and qualitative* comparison on consensus latencies with several caveats.

We have noted that a leader-based protocol also requires opportunistic protocols for message dissemination and a $\diamondsuit QC$ equivalent to propagate the decision which must reach all operative members. Hence, one could take a view that the time taken for completing a given round and that for total quiescence after the first member has decided will be comparable between the two protocols.

Therefore, assuming that no crashes occur, a leader based protocol will take exactly one round for decision if STO is chosen to be sufficiently large. This means that our protocol, taking 3.2 - 2.2 rounds (on an average with minor variations) to decide for the densities considered, can take up to approximately 3 times the time taken by a leader-based protocol to reach decision. If the chosen STO causes 2 unwarranted leader changes, then both the protocols would perform nearly identically. If the chosen STO causes 3 or more unwarranted leader changes, our protocol would certainly achieve total quiescence faster.

To summarize, the question of whether the leader-based approach to consensus is better suited to sparse MANETs than the randomized approach depends primarily on how *reliably* the right STO is found and *maintained* over the lifetime of a sparse MANET. It is mainly to avoid the difficulties in reliable estimation of STO, a randomized consensus protocol was first proposed (in 1983) [BO83] in the context of asynchronous communication systems where, unlike in LAN or always-connected

MANETs, the bound on prevailing communication latencies cannot be reliably estimated. The latter aspect is modeled in [BO83] as the communication network being controlled by an unknown adversary who keeps the latencies finite but controls them in an arbitrary manner.

6.4 Summary

The consensus protocol introduced in this chapter was derived from the *EMR* protocol. The derived protocol makes use of $\diamondsuit RC$ and $\diamondsuit QC$ protocols. The $\diamondsuit QC$ protocol is used only to disseminate decisions. So the $\diamondsuit RC$ is used excessively by members to exchange estimates in all phases/rounds until reaching a decision.

While members in *EMR* exchange each other's estimates (*est* values), members in this protocol send an extra value; the input candidate for the random selection process *cand*. A member N_i uses $VBag_i$ to maintain received distinct *cand* values in the current phase. At the end of each phase, $cand_i$ is randomly chosen from $VBag_i$ to be sent with est_i. So this $cand_i$ will be the input candidate for the random selection to be held at the end of next phase. When two mebers N_i and N_j randomly select the same value for their *cand* values, $VBag$s of operative members in the next phase will decrease in size. So repeating the random selections, frequently enough, will result in reducing the size of $VBag$s of operative members until these $VBag$s eventually become identical singleton sets. This makes operative members have the same *cand* after any 'random' selection. So mebers can decide on this *cand* value. We refere the reader to our work in [AE10] for the proof.

This protocol copes with the challenges which might arise from using the $\diamondsuit RC$ protocol. These challneges are mainly expressed by the deadlock which might occur when $n - 2f \leq \frac{n}{2}$. This deadlock is encountered because $\diamondsuit RC$ guarantess that at least $n-2f$ correct members receive any multicast and the consensus protocol needs members to receive from a majority. So when $n-2f$ is less thatn majority, members might not be able to proceed their rounds. To avoid the deadlock, operative members repeat their $\diamondsuit RC\_mcasting$ of (r, ph) if they find themselves being unable to receive (r, ph) messages from a majority of members. The repetition of $\diamondsuit RC_mcast$ing of (r, ph) messages is done in a reasonable manner to avoid any overhead. Note also that any repeated (r, ph) messages are treated as distinct mulitcast messages by $\diamondsuit RC$.

Simulations show that the average number of rounds, taken to recach decision, is amazingly small; it is between 2.2 and 3.2 for all used densities. The latency for this protocol decreases dramatically when the node density increases; lower densities incur long time to reach the consnesus (as expected). Moreover, increasing the nodes maximum speed causes the latency to be decreased in particular in lower densities.

The node density has a significant influence on the data packet overhead as higher densities incur less data packet overhead. While node speeds show no influence on the data packet overhead.

The control packet overhead shows a similar trend as the latencies, suggesting that the longer it takes to reach consensus, the more control packets are being expended.

Chapter 7

Summary and Conclusions

Consensus being a fundamental problem in distributed computing, it has been addressed in MANETs, often in association with addressing another MANET-centric concern: [WCR09] and [WCYR07] solve consensus together with clustering and scalability issues, [CDG+05] with packet collisions, [GT07] and [CSS04] in the context of participants' identities not known initially. This thesis considers consensus for sparse MANETs where flooding [HOTV99] or mesh/tree based routing [VOT06] cannot guarantee multicast coverage necessary for known solutions to be simply deployed. So a consensus module has been built for supporting collaborations in MANETs. This module has been shown to operate even in sparse and highly mobile MANETs, and expanded simulations confirm that the overheads incurred are not too high to be feasible in MANETs.

This chapter summarizes the contributions made in this dissertation and illustrates some routes of future research.

7.1 Summary

Chapter 2 presented the fundamental related work on multicast protocols and solving the consensus problem in MANETs. A brief comparison between the performance of some existed categories of multicast protocols was carried out. It was also observed that few consensus protocols have been subject to performance evaluation. These protocols were introduced briefly and an explanation about the weakness of each of them was presented.

In chapter 3, a system model for MANETs was presented. The system model accurately defines the requirements on the network conncectivity, the multicast pro-

tocols which will be used for solving the consensus and the approach to the consensus. The requirement on the network connectivity was intended to be as minimal as possible, and essentially only requires operative members not to be isolated permanently. The principles of the required multicast protocols were defined in details. Finally, the approach to the consensus was introduced.

Chapter 4 presented a derived multicast protocol called Encounter Gossip Multicast (*EGM*, for short). This protocol has been used, as an optional service, by the required multicast protocols. *EGM* is also a $\diamondsuit R$ protocol because it ensures that there is a time after which operative members relinquish a multicast message. Moreover, *EGM* is a network topology-independent which makes it work without the need to store any information about the network topology. An optimized version of *EGM* was introduced in the same chapter. The simulations showed that the optimized *EGM* gives a high coverage with affordable cost and delay.

In chapter 5, two new multicast protocols, $\diamondsuit RC$ and $\diamondsuit QC$, required for solving consensus were presented. In addition to another protocol used as underlying service to build and maintain (i) the H-hop group neighbourhood list (when $H \geq 1$) and routes to each group neighbour (when $H > 1$), and (ii) 1-hop neighbourhood list which includes members and non-members. These neighbourhood lists are needed by the required multicast protocols. $\diamondsuit RC$ was explained in details by using two implementations; one by using *EGM* as an added service and another one without using any optional services. Simulations have proved that $\diamondsuit RC$ which used *EGM* as an added service performs better. $\diamondsuit QC$ was introduced using *EGM* as an added service. Intesive simulations for these protocols showed that they peform well in all scenarios with extra cost in sparse ones which is an expected behaviour.

Chapter 6 presented the consensus protocol which is derived from [EMR01]. The derived protocol was implemented on top of $\diamondsuit RC$ and $\diamondsuit QC$ protocols. The use of $\diamondsuit RC$ was kept to maximum so members use it to exchange their estimates in all phases/rounds until reaching a decision. The $\diamondsuit QC$ protocol was used only to send decisions. All the challenges which might be imposed by using $\diamondsuit RC$ were addressed and the derived consensus protocol can cope with these challenges. Simulations showed that the average number of rounds taken to recach decision was small in all densities (maximum was 3.2). Moreover, the time and packet overhead were not too high to be reasonable in MANETs.

7.2 Conclusion

This thesis has built a consensus module that supports a wide range of MANETs scenarios including sparse ones, and further that implementing this module deos not incur too high time and packet overhead to be feasible. It has done so by first defining the required condition on the network connectivity for the consensus to be solvable. Then, two multicast protocols, essential for consensus, have been developed before deriving the consensus protocol which addresses all challenges arised due to using the developed multicast protocols. Finally, extended simulations have been displayed which show that these protocols do not incur excessive time and transmission overhead.

7.3 Future Work

In this thesis we have shown the performance of our consensus protocol in a range of scenarios. Yet it would be preferable to provide a more in depth compariosn study with other protocols. So as a future work, we would like to investigate how our protocol performance compares with others, such as [WCR09] and [WCYR07], when the network becomes moderately dense.

The $\diamondsuit RC$ and $\diamondsuit QC$ protocols use an optional $\diamondsuit R$ service. In this thesis we have compared the pefromance of $\diamondsuit RC$ protocol using two implementations; one implementation using *EGM* protocol as an added $\diamondsuit R$ service and the another implementation without adding any $\diamondsuit R$ optional services. However, it would be prudent to add more comaprison studies by using other $\diamondsuit R$ protocols as added services and compare the performance.

The H-hop group neighbourhood list is used by both $\diamondsuit RC$ and $\diamondsuit QC$ protocols. The value of H which has been used throughout simulations in this thesis is 2. It was shown in section 5.7.3 that using $H = 2$ has influence on the latency incurred by $\diamondsuit RC$. However, the influence of the value of H can be investigated more by using different values. So as a future work, we would like to increase the value of H to compare any results with our cuurent ones.

Group members periodically send *Ghello* packets to update the group neighbourhood list. So, throughout this thesis, members send *Ghello* packets every 2 seconds interval. Choosing a small value of this interval might result in a big control packet

overhead because members will send *Ghello* packets more frequently. In contrast, choosing a big value might result in using invalid routes because members will try to update their routes less frequently, so some changes might occur on routes without members being aware of these changes. The performance study in sction 5.7.3 has proved that members send only few data packets before the realization of a message m. So this gives the implicastion that there were no loss in the transmitted data packets which means that the used routes were, mostly, valid. However, we would like to know more about the effect of this interval by using different values. So a potential future work would be to implement our protocols using different values of this interval in particular values greater than 2.

The simulation runs, though carried out in a cluster, were quite time-consuming because of low density values used. For density values smaller than 1.6, our protocol will work if the liveness condition is met but one should expect latencies to be in the order of hours, in particular, with slow moving nodes. So as a future work, we would like to run our protocol in lower densities than the used ones.

Bibliography

[ACT00] Marcos Kawazoe Aguilera., Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM J. Comput.*, 29:2040–2073, April 2000.

[AE10] Khaled Alekeish and Paul Ezhilchelvan. Consensus in Sparse, Mobile Ad hoc Networks. In *Technical Report, CS-TR-1208*, 2010.

[Asp10] Mikael Asplund. *Disconnected Discoveries: Availability Studies in Partitioned Networks*. PhD thesis, 2010.

[BEV06] François Bonnet, Paul Ezhilchelvan, and Einar Vollset. Quiescent consensus in mobile ad-hoc networks using eventually storage-free broadcasts. In *Proc. of 21st ACM Symposium on Applied Computing (SAC)*, pages 670–674, 2006.

[BHvR05] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: an efficient approach to simulation using virtual machines. *Softw., Pract. Exper.*, 35(6):539–576, 2005.

[BLMT98] E. Bommaiah, M. Liu, A. Toh McAuley, and R. Talpade. Amroute: Adhoc multicasting routing protocol. In *draft-talpade-manet-amroute-00.txt*. IETF, manet, 1998.

[BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[BPS08] Fatemeh Borran, Ravi Prakash, and Andre Schiper. Extending Paxos/LastVoting with an Adequate Communication Layer for Wireless Ad Hoc Networks. In *Proc. of 27th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 227–236, 2008.

[CDG+05] Gregory Chockler, Murat Demirbas, Seth Gilbert, Calvin Newport, and Tina Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proc. of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 197–206, 2005.

[CEM09] D.E. Cooper, P. Ezhilchelvan, and I. Mitrani. Encounter-based message propagation in mobile ad-hoc networks. *Ad Hoc Networks*, 7(7):1271 – 1284, 2009.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[CSS04] D. Cavin, Y. Sasson, and A. Schiper. Consensus with unknown participants or fundamental self-organization. In *Proc. of the 3rd Int. Conf. on ADHOC-NOW*, pages 135–148, Vancouver, 2004.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[CTA02] W. Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DN05] H. Dhillon and H.Q. Ngo. Cqmp: a mesh-based multicast routing protocol with consolidated query packets. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 4, pages 2168 – 2174 Vol. 4, 13-17 2005.

[EMR01] Paul Ezhilchelvan, Achour Mostefaoui, and Michel Raynal. Randomized multivalued consensus. In *Proc. of the 4th Int. Symp. on Object-Oriented Real-Time Computing (ISORC01)*, pages 195–200, 2001.

[Fal03] Kevin Fall. A delay tolerant network architecture for challenged internets. In *ACM SIGCOMM*, pages 27–34, August, 2003.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[GLAM99] J. J. Garcia-Luna-Aceves and Ewerton L. Madruga. The core-assisted mesh protocol. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 17(8):1380–1394, 1999.

[GS97] Rachid Guerraoui and Andre Schiper. Consensus: The big misunderstanding. In *FTDCS '97: Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.

[GT07] F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Proc. of the International Conference on Dependable Systems and Networks, (DCCS Track)*, pages 82–91, Edinburgh, UK, 2007.

[HGR04] Radu Handorean, Christopher Gill, and Gruia-Catalin Roman. Accommodating transient connectivity in ad hoc and mobile settings. *Lecture Notes in Computer Science*, 3001:305–322, March 2004.

[HOTV99] Christopher Ho, Katia Obraczka, Gene Tsudik, and Kumar Viswanath. Flooding for reliable multicast in multi-hop ad hoc networks. In *Proc. of the 3rd ACM workshop on Discrete algorithms and methods for mobile computing and communications (DIALM)*, pages 64–71, 1999.

[jLBrP03] Sung ju Lee, Elizabeth M. Belding-royer, and Charles E. Perkins. Scalability study of the ad hoc on-demand distance vector routing protocol. *International Journal of Network Management*, 13:97–114, 2003.

[KDPH05] Dimitrios Koutsonikolas, Saumitra M. Das, Himabindu Pucha, and Y. Charlie Hu. On optimal ttl sequence-based route discovery in manets. In *Proceedings of the Second International Workshop on Wireless Ad Hoc Networking - Volume 09*, ICDCSW '05, pages 923–929, Washington, DC, USA, 2005. IEEE Computer Society.

[KEJ05] B. Kaliaperumal, A. Ebenezer, and Jeyakumar. Adaptive core based scalable multicasting networks. In *INDICON, 2005 Annual IEEE*, pages 198 – 202, 11-13 2005.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LFA04] Mikel Larrea, Antonio Fernandez, and Sergio Arevalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Transactions on Computers*, 53:815–828, 2004.

[LGC99] Sung-Ju Lee, M. Gerla, and Ching-Chuan Chiang. On-demand multicast routing protocol. In *Wireless Communications and Networking Conference, 1999. WCNC. 1999 IEEE*, pages 1298–1302 vol.3, 1999.

[LSH+00] Sung-Ju Lee, W. Su, J. Hsu, M. Gerla, and R. Bagrodia. A performance comparison study of ad hoc wireless multicast protocols. volume 2, pages 565 –574 vol.2, 2000.

[RP99] Elizabeth M. Royer and Charles E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 207–218, New York, NY, USA, 1999. ACM.

[VOT06] Kumar Viswanath, Katia Obraczka, and Gene Tsudik. Exploring mesh and tree-based multicast routing protocols for manets. *IEEE Transactions on Mobile Computing*, 5(1):28–42, 2006.

[vRMH98] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-based failure detection service. In N. Davies, K. Raymond, and J. Seitz, editors, *Proc. of the Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 55–70, September 1998.

[WCR09] Weigang Wu, Jiannong Cao, and Michel Raynal. Eventual clusterer: A modular approach to designing hierarchical consensus protocols in manets. *IEEE TPDS*, 20:753–765, 2009.

[WCYR07] Weigang Wu, Jiannong Cao, Jin Yang, and Michel Raynal. Design and performance evaluation of efficient consensus protocols for mobile ad hoc networks. *IEEE Trans. on Computers*, 56:1055–1070, 2007.

[WTa98] C. Wu, Y. Tay, and C. Toh and. Ad hoc multicast routing protocol utilizing increasing id-numbers (amris). In *draft-ietf-manet-amris-spec-00.txt*. IETF, manet, 1998.

[XC09] Yong Xi and Mooi Choo Chuah. An encounter-based multicast scheme for disruption tolerant networks. *Comput. Commun.*, 32(16):1742–1756, 2009.

[Zha06] Zhensheng Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges. *IEEE Communications Surveys and Tutorials*, 8(1-4):24–37, 2006.