A distributed convergecast algorithm for dynamic mobile networks

Aymeric Agon-Rambosson, Jonathan Lejeune, Julien Sopena and Pierre Sens
Sorbonne Université, CNRS, LIP6, DELYS
F-75005 Paris, France
firstname.lastname@lip6.fr

***Abstract*—Some applications, like round-based consensus algorithms, require all the nodes from a system to send a message to the same node (the leader) at the same time. In a Mobile Ad-Hoc Network (MANET), this situation is likely to cause collisions and the loss of the messages converging to the leader. The loss of messages is critical in such a situation, since the leader needs to receive a quorum of messages to make a decision. This pattern of communications, called *convergecast*, can be trivially implemented with a unicast primitive. However, we show that a popular MANET unicast algorithm like Optimized Link State Routing (OLSR) loses a lot of messages, even in the presence of MAC-level collision avoidance mechanisms like CSMA/CA. We propose a new convergecast algorithm that locally schedules answers to a query in a fully distributed manner, in order to avoid their colliding with each other, and that aggregates these answers in order to further decrease the probability of collisions. We show that our algorithm creates far fewer collisions and retries than OLSR, allowing applications like consensus algorithms to reach their quorum sooner.**

***Index Terms*—MANET, Convergecast, Message aggregation, OLSR, Consensus, Paxos**

I. Introduction

A Mobile Ad-Hoc Network (MANET) consists of heterogeneous mobile nodes communicating wirelessly directly with each other. This network paradigm developed at the end of the 1990s can accurately model edge computing environments, meshnets, wireless sensor networks, as well as networking situations consistent with IoT applications.

Since this paradigm assumes no preexisting network infrastructure, the nodes have to act as relays for packets not intended for them. Wireless communications are subject to faults, particularly collisions: a node inside the intersection of the covered areas of two other nodes transmitting at the same time will receive neither message correctly. In the case of wireless communications, these collisions are impossible to detect, impractical to avoid, and costly to address [1]. This problem is particularly acute for the *convergecast* operation, which is the communication pattern where a (generally large) subset of nodes of the network all send a (different) message to the same node. This operation creates a lot of situations in which a node is likely to receive messages from several of its neighbors at the same time, without the sending neighbors being capable of hearing each other, thus defeating any potential Channel Sensing Multiple Access with Collision Avoidance (CSMA/CA) [2] transmission delaying mechanism present at MAC level.

Convergecast can be seen as a special case of *unicast* (the operation by which a node sends a message to another node), in which the destination is the same for everyone, and the messages are all sent at roughly the same time. It is therefore possible to trivially implement convergecast on top of a unicast algorithm. We show however that when we try to implement convergecast on top of Optimized Link State Routing (OLSR) [3, 4, 5], a popular unicast algorithm for MANETs, we lose a lot of messages due to collisions. The retry mechanisms included in CSMA/CA allow to partly alleviate the problem, but at a huge cost in extra messages.

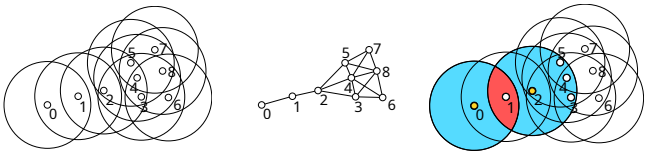
Convergecast has been an interest of the Wireless Sensor Networks (WSN) [6], in which a set of small, power-constrained nodes must regularly forward information to a base station, to which they are not necessarily directly connected. WSNs assume a fixed topology, perfect information on the network and synchronized clocks. For this reason, papers like [7, 8, 9, 10] present algorithms that are unsuitable for MANETs.

Contributions. We address this problem by proposing a distributed convergecast algorithm that strives to be:

- *effective*, by achieving a high reception rate at the destination,
- *efficient*, diminishing the amount of bytes sent and received by each antenna as much as possible, thereby decreasing energy consumption,
- *low-latency*, by drastically reducing the amount of retries needed.

This algorithm is designed for the query-response pattern, in which a node will broadcast a request to the entire network, and every other node will have to respond. This pattern of communications is typical of round-based consensus algorithms like [11] or other information gathering applications. Our algorithm is built on two mechanisms, namely a distributed and local scheduling of the response messages, and the aggregation of those response messages on the way back to the requester. We target generic MANETs, where no global information on the network is available to any node, and possible mobility makes long-term centralized construction of spanning trees impractical. We have also implemented the algorithm in the OMNeT++/INET discrete event simulator [12], in order to

This paper is funded by the French ANR project SeMaFoR



(a) Coverage areas (b) Graph representation (c) Collisions when 0 and 2 send simultaneously

Fig. 1: Because of constant transmission range, the system can be represented as an undirected graph.

provide a quantitative evaluation of various metrics against OLSR (that we have implemented as well).

This paper is organized as follows. We present our hypotheses regarding the system, the model of communications and the motivation (§II). We then present the algorithm's prerequisites and formal specification (§III). We then present a quantitative evaluation of our algorithm on simulated graphs using a full network stack simulation in OMNeT++/INET [12] (§IV). Finally, we briefly present related works of the literature from the WSN (Wireless Sensor Networks) paradigm (§V) and conclude (§VI).

II. Model, background and motivations

We start by specifying the system and communication models. We then describe the MAC 802.11 protocol, and the specification and motivation of the convergecast protocol for MANETs.

A. System and communication models

We consider a constant set N of uniquely identifiable nodes n_0, n_1, ..., n_{N-1} in a 2-dimensional square. We assume no availability of GPS information on the devices. Nodes communicate via omnidirectional wireless transceivers with fixed transmission T and reception R ranges that are identical and large enough to guarantee network connectivity.

The *coverage area* of a node is therefore given by the circle centered on the node of radius T (Fig. 1a). When the node emits a frame, all other nodes within that circle receive the signal with a probability 1, while all nodes outside of the circle receive the signal with probability 0. Hence, the cost to transmit a message to one's immediate *neighborhood* (the nodes within transmission range) is independent of the size of that neighborhood. Because of physical limitations of radiotransmitters, nodes are not able to send and receive simultaneously. The system can therefore be modeled as a *geometric graph*: whenever a node u is part of the neighborhood of node v, then the opposite is also true, and this relationship can be represented by an undirected edge connecting u and v (Fig. 1b). Now, whenever a node sends a message, all *adjacent* nodes in the graph can be considered to have received it, with one notable exception: as mentioned earlier, a node located in the intersection of the coverage areas of two distinct nodes that transmit a frame simultaneously will receive neither signal correctly: this phenomenon is called *collision* (Fig. 1c). Collisions are impossible to detect by the sender [2], and must therefore be avoided by some Medium Access Control protocol.

B. IEEE 802.11 MAC protocol

The role of the MAC layer of the IEEE 802.11 protocol is to schedule the use of the wireless medium (in our case, the carrier radio wave) in order to avoid collisions. In this protocol, nodes that wish to transmit a frame must do so according to a protocol known in the literature as CSMA/CA (Carrier Sensing Multiple Access with Collision Avoidance): they first listen to the carrier to check that no other node is already transmitting. More specifically, if the node detects no signal for a period of time called DIFS (*DCF InterFrame Spacing*), it sends the frame. Otherwise, the node creates a random counter called *Contention Window* (according to a uniform distribution between 0 and a standard-specified quantity called *CW*), and decrements it each time it senses that there has been no signal for a period of time called *SlotTime*. The frame is sent when the counter reaches 0. This protocol avoids collisions, but only when the potential transmitters can hear each other.

On top of that, the protocol **optionally** allows the sender node to ask the nodes that receive the frame correctly to wait a period of time called *SIFS* (*Short InterFrame Spacing*), and then send an acknowledgement frame. If the sender does not receive the acknowledgement frame after a specific time period called *AckTimeout*, or if the acknowledgement frame is negative (NACK), the sender retries the whole process with a larger *Contention Window*. This *Contention Window* grows exponentially in the amount of retries, and there is a maximum number of retries.

C. Optimized Link State Routing

As previously mentioned, convergecast can be trivially implemented on top of unicast. The OLSR (Optimized Link State Routing) protocol is a popular unicast algorithm for MANETs. In this algorithm, each node starts by broadcasting **locally** (i.e., only to its neighbors, who do not retransmit) its list of neighbors in a HELLO message. Thus, every node in the system eventually knows the list of its 1-hop and (2)-hop neighbors, and all the links of each of its 1-hop neighbors. Each node then selects among its neighborhood a set of relays according to the MPR algorithm [13], and marks this set in the HELLO message as well. Consequently, every node eventually knows by which of its neighbors it was selected as MPR relay. Then each node who was selected as MPR by at least one of its neighbors must broadcast **globally** to the system the set of neighbors by which it has been selected, in a message called TC (Topology Control). Notice that each MPR selectee is a last-hop node to each of its MPR selectors. Hence, since each node must choose some MPR relays, every node is capable of building a *routing table* for every node in the system, using this information and some shortest-path algorithm. The routes provided by these routing tables are guaranteed to be a shortest path.

This algorithm is very suitable for MANETs because it greatly optimizes the amount of data needed to be exchanged in order to compute the routing tables. It is in fact the optimized version of the Link State Routing protocol used in classical infrastructure networks [1]. Nonetheless, as we will see now, even well-optimized unicast algorithms like OLSR are ill-suited to provide a convergecast primitive.

D. Motivation

Let us illustrate this last point with a simple experiment that we carried out in the discrete event simulator OMNeT++ on a random geometric graph. Let us assume that some node of the network wishes to send a request to all the other nodes. This request will be *broadcast* using some MANET broadcast algorithm like MPR. The other nodes, upon reception of the request, have to send a response back to the requester, to imitate a round-based consensus algorithm. In our experiment, this response will be sent using OLSR. Fig. 2a shows one graph, with the requester marked in yellow, the nodes whose answer has not arrived at the end of the experiment in red, and the edges on which response messages were lost are marked in red as well. We see that very few response messages (14 including the response from the requester) actually arrive to the requester, because of losses **on the way back** (every node received the request message): the responses collided with each other on the way back to the requester. This behavior is confirmed when the experiment is run on a large sample of graphs of varying average degree, as we can see on Fig. 2b: few request messages are lost (less than 5%), but the rate of received responses is between 30% for low-degree graphs, and less than 10% for high-degree ones. The CSMA/CA random delay mechanism that was active during the experiment did not help much, since the transmitters of the colliding messages frequently did not hear each other.

As we see, the convergecast operation is particularly susceptible to collisions, and the random delay implemented by the MAC layer is not particularly helpful. We therefore propose a new algorithm, Distributed Convergecast (DC), that strives to achieve better reception rate, without increasing the amount of bytes sent and the latency.

III. Algorithm

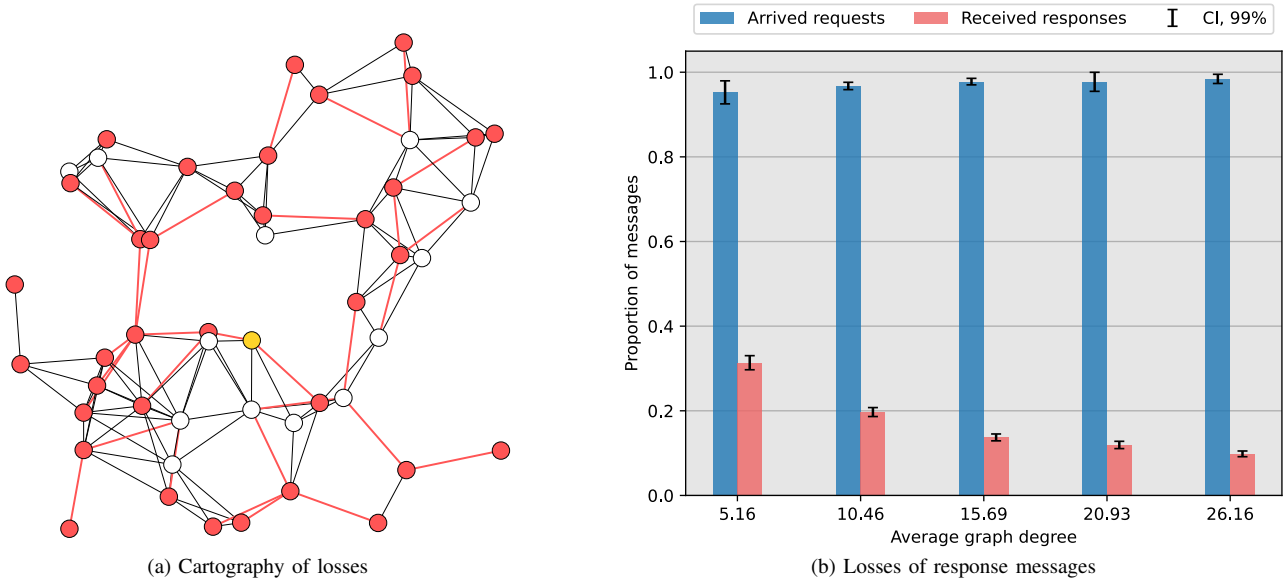
The goal of a MANET convergecast protocol is to support the sending of messages from a subset of nodes of the network to the same node. Any subset of nodes (in particular, the entire network) is liable to send a message at any time. The goal is to achieve a good *reception rate* (i.e., the proportion of messages received by the common destination), while minimising the amount of bytes sent (thus minimising the energy consumption) and the latency.

The nodes are assumed to have no previous information whatsoever on the network. However, we assume that the convergecast operation to a specific destination follows a broadcast by that destination. Hence, we assume that a MANET broadcast protocol is implemented in the network (in our case, MPR). A complete description of the MPR broadcast algorithm is out of the scope of this paper.

We will briefly explain the principle of the algorithm before providing a complete formal specification. The idea is to use information about the immediate neighborhood (since it is needed and computed by the broadcast algorithm anyway) to calculate deterministic delays, in order to **locally** schedule the responses to the node who transmitted the request, therefore ensuring that they cannot collide with each other.

A. Prerequisites

Our algorithm requires to keep and transmit some information during the broadcast phase. Broadcast algorithms, even the



(a) Cartography of losses

(b) Losses of response messages

Fig. 2: Many more messages are lost on the way back

simplest ones [14, 15, 16], keep a memory of retransmitted and delivered messages, in order to avoid retransmitting multiple occurrences of the same message, or delivering it multiple times to the applicative layer (some broadcast specifications forbid multiple deliveries of messages). This memory stores a unique identifier of the message (some tuple made of the identifier of the broadcast sender, and a sequence number). For our purposes here, we need this memory to also include:

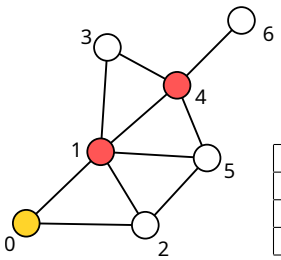
- the direct transmitter of the message (the "father"), inferable from the MAC header of the message;
- the other recipients of the message ("the brothers"), minus those (the "uncles") that had already received it from the transmitter of the transmitter (the "grandfather"). This is inferred from the neighborhood information, and some extra information to the MPR header;
- the (local) time of the delivery;

In the case this message is to be retransmitted by the node, we also need:

- all the neighbors this message will be retransmitted to (the "sons"), minus those that are already neighbors of the transmitter (and can thus be assumed to have received it already). This is also inferred from the neighborhood information, and some extra information contained in the MPR header;
- all the designated relays of the message, computed with the MPR heuristic [17]. This set is a subset of the former, and is included in the MPR header anyway;
- the (local) time of the retransmission.

Some of this information is needed by the MPR algorithm anyway for the broadcast to function correctly, and is therefore already part of the MPR header of the request message. However, our algorithm requires the following information to be included in it as well (we assume a request message sent from node u to node v):

- The node that transmitted the message to u (the "father" of u, and grandfather of v);
- The nodes that are neighbors of u's father, minus those that are already neighbors of u's grandfather (u's "uncles"), and minus u's father itself. Those nodes are the "brothers" of u.



(a) Graph disposition. Source in yellow, retransmitters in red

Message	Father	Brothers
$0 \to \langle 1, 2 \rangle$		$\langle\rangle$
$1 \to \langle 3, 4, 5 \rangle$	0	$\langle 2 \rangle$
$4 \to \langle 6 \rangle$	1	$\langle 3, 5 \rangle$

(b) Extra content in the MPR header

Fig. 3: Information transmitted during the request phase

We can see an example of how this extra information is transmitted on Fig. 3b. Notice that the information regarding one's father and brothers need only be transmitted on one hop. For instance, when 4 receives the message from 1, he can remove the information regarding 1's father and brothers, and replace it with its own. So the size of the MPR header does not grow with the number of hops, and is proportional to the average degree of the graph. This is the only extra information we need transmitted by the broadcast primitive.

B. Deterministic delay mechanism

The first part of our algorithm is the calculation of a deterministic delay for the response.

The idea is that whenever a node u receives a request message m, it knows the set of nodes that have received m from the same transmission (and are therefore expected to answer to it). We will call this set the **contention set** (which includes u). This set can be totally ordered, and u has a index i in this set. The idea is to allocate to each member of the contention set a non-overlapping timeslot according to its index, thereby creating a local and distributed by design schedule of the responses.

The timeslots need to be dimensioned carefully (too small timeslots will lead to collisions, too large ones will lead to latency degradation), and must be calculable independently by each node in the contention set. For this, we first assume that the bitrate of the antennae are fixed and known, which is not a particularly strong requirement. We also assume that the response messages to a specific request have the same size. Those two values allow us to calculate the time needed by the other nodes to send their message.

However, we also need to consider the randomization introduced by CSMA/CA. We know how CSMA/CA calculates the delay, and the various parameters mentioned earlier in §II-B (DIFS, SlotTime, CW) are fixed by the standard.

Thus, we can calculate an upper bound on the amount of time CSMA/CA adds to the transmission time, assuming that no potentially conflicting transmissions occur at the same time. For instance, on a transmission without retries (a broadcast, for instance), we have the following upper bound:

$$\text{Bound}_{\text{NR}} = \frac{M}{B} + \text{CW}_1 * \text{SlotTime} \tag{1}$$

With M being the message size, B the antenna bitrate, and CW_1 the maximal value of the random contention window.

When the optional retry mechanism is activated, calculating an upper bound becomes:

$$\text{Bound}_{\text{R}} = \sum_{r=1}^{\text{MaxRetries}} \left(\frac{M}{B} + \text{CW}_r * \text{SlotTime} + \text{AckTimeout} \right) \tag{2}$$

The *AckTimeout* and *MaxRetries* parameters are determined by the standard, but CW_r increases exponentially in the amount of retries:

$$\text{CW}_r = min(\text{CW}_{max}, \text{CW}_{min} * 2^{r-1}) \tag{3}$$

With CW_{max} and CW_{min} being determined by the standard, and r being the index of the retry (beginning at 1). The response timeslot for a node can be therefore be calculated like this:

$$\text{ResponseSlot}_i = \Big[\text{RT}_u + \text{RA} * \text{Bound}_{\text{NR}} + (i-1) * \text{Bound}_{\text{R}}; \\ \text{RT}_u + \text{RA} * \text{Bound}_{\text{NR}} + i * \text{Bound}_{\text{R}}\Big] \tag{4}$$

With i being the index of the node in the total order of the contention set, RT_u being the time (local to u) at which the request was received, and RA being the number of relays that will retransmit. A node that wishes to transmit a response to the node that it received the request from has to do it within that timeframe.

Due to application-specific reasons, a node might not be able to do so. For this reason, if the timeslot has already started when the response is ready to be sent, then the node must answer after the last timeslot is finished, that is after:

$$\text{EndResponseSlots}_u = \text{RT}_u + \text{RA} * \text{Bound}_{\text{NR}} + S * \text{Bound}_{\text{R}} \tag{5}$$

With S being the size of the contention set. Notice that if multiple nodes miss their slot, they will potentially transmit at the same time at the end of the last timeslot. The purpose of our mechanism is not to guarantee the total absence of collisions, but merely to make them less likely, which is good enough, as we will see later.

C. Message aggregation mechanism

Since the messages converge from the whole network to the requester, it makes sense to aggregate them on the way back in order to decrease the amount of bytes sent, and to dramatically decrease the amount of messages transiting in the network at a given time, which helps avoiding collisions.

A node retransmitting a broadcast message knows the subset of its neighbors that will have to answer to it. It is the set of its 1-hop neighbors, minus the message transmitter and its neighbors. This set will be called the **aggregation set**. With this set, we can implement a simple aggregation mechanism that will aggregate and delay all response messages received between the time of the broadcast retransmission and the time when the last node of the aggregation set is supposed to have finished answering. We dimension the aggregation window at node u proportionally to its aggregation set:

$$\text{AW}_u = \Big[\text{RetrT}_u + \text{RA} * \text{Bound}_{\text{NR}}; \\ \text{RetrT}_u + \text{RA} * \text{Bound}_{\text{NR}} + S * \text{Bound}_{\text{R}}\Big] \tag{6}$$

With $Retr_u$ the time at which u has finished retransmitting, and S the size of the contention set of the sons of u. Notice also that the aggregation does not wait for late nodes.

We can see on Fig. 4 that the response slots for nodes in the same contention set do not overlap: RT_4 designates the time at which 4 receives the request from 1, $RetrS(4)$ the timeslot during which 4 is supposed to retransmit the request to 6, $RS_i(u)$ the response slot of index i attributed to node u (according to 4), and ERS_4 the end of all the response slots according to 4. Since 3, 4 and 5 all receive the request from 1 at the same time, RT_4, RT_3 or RT_5 mean the same time, regardless of any clock drift or offset. Thus, 3, 4 and 5 always agree on the time of their respective response slots. On top of that, since $RetrT_1$ and RT_4 mean the same time (4 finishes receiving the message from 1 at the same time 1 finishes sending it to 4), the aggregation window of 1 is consistent with the response slots of 3, 4 and 5.

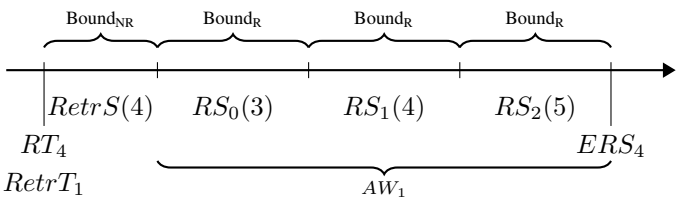


Fig. 4: Response timeslots for 3, 4 and 5, according to 4 and aggregation window of 1 (same topology as Fig. 3)

D. Algorithm specification

We recall that the convergecast follows immediately a broadcast phase that uses the MPR algorithm. During this broadcast phase, according to §III-A, the request message R is stored in the memory Mem. This request message contains, among the other things mentioned earlier, the transmitter of the request ($R.Tr$) and the time of its delivery ($R.T$).

The header of the response message M must contain at least the address of the requester ($M.S$), i.e., the destination of the response, and the sequence number of the request ($M.N$).

Our algorithm considers two cases:

- The message M is received from the upper layer (i.e., the node wishes to send a response message using convergecast).
- The message M is received from the lower layer (i.e., some other node sent a response to the current node, who must retransmit or deliver it).

The first case is described by algorithm 1: Upon reception of a response message from the upper layer, we search the memory of requests Mem defined in §III-A for the request corresponding to the response (line 2), according to the sequence number $M.N$ and request source $M.S$ specified in the response message. If the request cannot be found, the message is dropped and the algorithm finishes (line 5). Otherwise, the response timeslot is calculated according to equation 4 (line 6). If it turns out the timeslot is already over, the end of the last timeslot is calculated according to equation 5 (line 8). If it turns out that the last timeslot has already passed, the message M is sent to the request transmitter $R.Tr$ immediately, and

Algorithm 1: Convergecast : upon reception of message M from upper layer

```
   Input:
       M : message
   /* Memory of requests                                   */
1  global variable (set of requests) Mem;
   /* Corresponding request                                */
2  local variable (request) R ⟵ getRequest(Mem, M.N, M.S);
3  if R = nil then
4  |  drop(M);
5  |  return;
   /* Obtain start of own timeslot                         */
6  local variable (float) D ⟵ getTSStart(M, R);
   /* If late, obtain end of last timeslot                 */
7  if R.T + D < currentTime() then
8  |  D ⟵ getLastTSEnd(M, R);
   /* If after end of last timeslot, send right
      away                                                 */
9  if R.T + D < currentTime() then
10 |  send(M, R.Tr);
11 |  return;
   /* Otherwise, delay send at the timeslot                */
12 delayedSend(M, D, R.Tr);
```

Algorithm 2: Convergecast: upon reception of message M from lower layer

```
   Input:
       M : message
   /* Memory of queries                                    */
1  global variable (set of requests) Mem;
   /* Own address                                          */
2  global variable (address) A;
   /* Set of aggregated messages                           */
3  global variable (set of messages) AS;
   /* Corresponding request                                */
4  local variable (request) R ⟵ getRequest(Mem, M.N, M.S);
5  if R = nil then
6  |  drop(M);
7  |  return;
8  if A = M.S then
9  |  deliver(M);
10 |  return;
   /* Obtain end of aggregation window                     */
11 local variable (float) D1 ⟵ getAWEnd(M, R);
   /* Obtain end of last timeslot                          */
12 local variable (float) D2 ⟵ getLastTSEnd(M, R);
   /* Obtain delay                                         */
13 local variable (float) D ⟵ max(D1, D2);
   /* If after delay, retransmit and end                   */
14 if R.T + D < currentTime() then
15 |  send(M, R.Tr);
16 |  return;
   /* Otherwise, aggregate and delay send                  */
17 M ⟵ addOrAggregate(AS, M.N, M.S, M);
18 delayedSendAndClean(M, D, R.Tr, AS);
```

the algorithm finishes (line 11). Otherwise, the message is sent to the request transmitter with the chosen delay D (line 12).

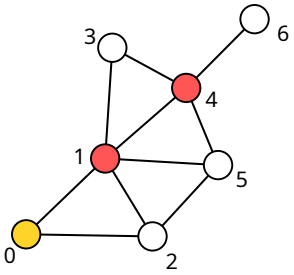
The second case is described by algorithm 2: Upon reception of a response message from the lower layer, we search the memory of request Mem for the request corresponding to the response (line 4). If the request cannot be found, the message is dropped and the algorithm finishes (line 7). Then, if the node A executing the algorithm is the intended recipient $M.S$ of the message, it is delivered to the upper layer and the algorithm terminates (line 10). Otherwise, the message must be forwarded to the transmitter of the request $R.Tr$. First, the aggregation window is obtained according to equation 6 (line 11), as well as the end of the last timeslot of the contention set of the node executing the algorithm, according to equation 5 (line 12). The delay is set to the maximum of these last two values (line 13). If this delay has already passed, the response message is retransmitted immediately to the transmitter of the corresponding request $R.Tr$ (line 16). Otherwise, the response message is aggregated with the others of the same aggregation window, and this set of aggregated messages is sent to the request transmitter with the chosen delay D (line 18).

Note that a node retransmits aggregated response messages necessarily after the end of the last timeslot of its contention set, thanks to line 13, and not within its own timeslot. This is because aggregated messages are larger than individual messages, and the timeslots have been calculated according to the size of individual messages, so the transmission of an aggregated message inside a timeslot would probably spill over the next timeslot.

E. Example

In order to better understand the execution of the algorithm, we will run it on the simple graph on Fig. 5. Node 0 makes the request, and every other node is supposed to answer. In order to simplify the example, it is assumed that the application is always able to produce its response in time for the corresponding timeslot. Node 0 starts by triggering a broadcast of the request using the MPR broadcast algorithm. It registers the empty set in its contention set (since it is the origin of the request and won't have to retransmit any answer), and its two sons 1 and 2 in its aggregation set. When 1 and 2 receive the transmission, they both put $\langle 1, 2\rangle$ in their contention sets: they are the neighbors of 0, and 0 itself received the request from noone else. As soon as 1 receives the request transmission, it forwards it, and puts $\langle 3, 4, 5\rangle$ in its aggregation set: it knows they are the only **new** recipients of the request. In the MPR header, it includes the information that 0 is its father and 2 its brother. When 3, 4 and 5 receive the request transmission from 1, they all put $\langle 3, 4, 5\rangle$ in their contention set: they know that 2 is their uncle not their brother.

At that time, 4 retransmits the request to 6 (and puts 6 in its aggregation set), but 1 concurrently responds to 0. Since 0 cannot hear 4, and 6 cannot hear 1, the messages cannot collide with each other. 6 responds to 4 straight after receiving the request. After 1 has finished responding to 0, 2 does the same. However, this last response can overlap with the response from 3 to 1. In this case, 1 can hear both 2 and 3, so the messages have a chance of colliding. On top of this, since 3 and 2 cannot



(a) Graph disposition. Source in yellow, retransmitters in red

Node	CS	AS
0	$\langle\rangle$	$\langle 1,2\rangle$
1	$\langle 1,2\rangle$	$\langle 3,4,5\rangle$
2	$\langle 1,2\rangle$	$\langle\rangle$
3	$\langle 3,4,5\rangle$	$\langle\rangle$
4	$\langle 3,4,5\rangle$	$\langle 6\rangle$
5	$\langle 3,4,5\rangle$	$\langle\rangle$
6	$\langle 6\rangle$	$\langle\rangle$

(b) Contention and aggregation sets

Time interval	Origin	Type	Dest.
[0, b]	0	Req	1,2
[b, 2b]	1	Req	3,4,5
[2b, 3b]	4	Req	6
[b + B, b + B + U]	1	Resp (1)	0
[b + B + U, b + B + 2U]	2	Resp (2)	0
[2b + B, 2b + B + U]	3	Resp (3)	1
[2b + B + U, 2b + B + 2U]	4	Resp (4)	1
[2b + B + 2U, 2b + B + 3U]	5	Resp (5)	1
[2b + B + 3U, 2b + B + 4U]	1	Resp (3,4,5)	0
[3b, 3b + U]	6	Resp (6)	4
[2b + B + 3U, 2b + B + 4U]	4	Resp (6)	1
[2b + 4U, 2b + 5U]	1	Resp (6)	0

(c) Produced schedule: b is the duration of a broadcast transmission, B is Bound_{NR} and U is Bound_{R}

Fig. 5: Algorithm walkthrough

hear each other, the random delay mechanism of CSMA/CA cannot be guaranteed to deterministically avoid this collision. After 3 has finished responding to 1, 4 does the same (but only with its own response, not the one it has received from 6). Then 5 does the same. The end of the timeslot of 5 coincides with the end of the aggregation window of 1, who retransmits the aggregated responses of 3,4 and 5 to 0. At the same time, 4 transmits the response of 6 to 1 (since 4 must wait for its last brother, 5, to have finished transmitting before transmitting responses from its own descendents, per line 13 of algorithm 2). When 1 gets 6's response from 4, it forwards it to 0. At that time, all the responses have reached 0, no more messages transit in the network for that specific request.

This algorithm avoids the most obvious collisions by using only local information: notice that brothers from a same father will always have the same contention set, itself identical to the aggregation set of their father. This means that brothers will never contend with each other as long as they can make their timeslot. This local and distributed scheduling also allows for non-colliding transmissions to take place concurrently. This algorithm does not propose to avoid all possible collisions. In fact, several such convergecast operations with different sinks could be taking place in the same MANET at the same time, and our algorithm offers no mechanism to avoid the collisions incurred by this. However, we will see in the evaluation that we are able to avoid many more of them than OLSR.

IV. Evaluation

A. Metrics

We evaluate DC according to the following metrics:

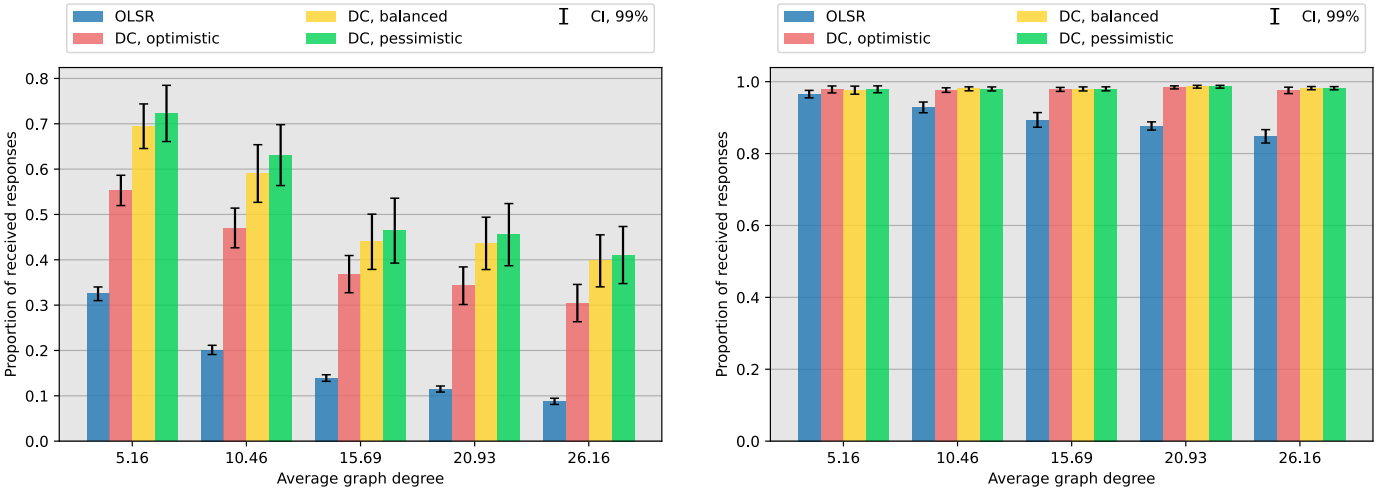
- The response rate, that is the amount of responses received by the requesting node divided by the amount of nodes in the network.
- The total amount of bytes sent in the network for the response messages. This metric measures the cost of the algorithm, indirectly the energy cost. Every response message is counted, in particular the retries caused by the MAC-level CSMA/CA mechanism. Thus, an algorithm that causes many collisions, and therefore relies a lot on those retries to be effective, will be very costly.
- The latency of the responses. Since there are multiple responses, this is not straightforward. We could define this latency as the mean of the latencies of all received responses, but this would mean not taking into account the responses that never arrive, and comparing the latencies despite potentially very different response rates. For this reason, we define the latency as the time required to receive a certain proportion of responses defined as a parameter. This will only be possible if the response rate is above this proportion.

We also want to evaluate the impact of the optional CSMA/CA retries on those various metrics. For this reason, we propose to evaluate our algorithm and OLSR with both settings. We determined earlier that the backoff of the retries is exponential, so we want to experiment with different hypotheses regarding the maximal amount of needed retries. Naturally, only our algorithm will be impacted by this setting. We propose the following values :

- An optimistic version, which sets Bound_{NR} assuming that no retries at all are needed, i.e. that transmissions succeed on the first attempt.
- A balanced version, which assumes that at most one retry is needed, i.e. that transmissions succeed on the second attempt at the latest.
- A pessimistic version, which assumes that all the retries are needed.

B. Graph generation

Since the expected gain of our algorithm depends on the structural properties of the graph, the way to generate the graphs matter. We generate random geometric graphs [18]: we place N nodes at random in a 500 x 500 meters square, according to a uniform distribution. We assume a range of radiotransmitters R of 100 meters. Two nodes are connected if, and only if, their Euclidean distance is less than 100 meters. Such a network is not necessarily connected, we therefore



(a) Without MAC-level retries

(b) With MAC-level retries

Fig. 6: Reception rate at requester

prune the generated networks that aren't. Since the value of N will have a direct influence on the average degree of the graph, and therefore on the probability of collisions, we propose to create 5 samples of 100 random geometric graphs with N being respectively 50, 100, 150, 200 and 250. In our simulated samples, this corresponds to an average node degree of respectively 5.16, 10.46, 15.69, 20.93 and 26.16.

C. Discrete event simulator

For a realistic simulation of the behavior of OLSR and DC on MANETs, we need an accurate modelisation of the WiFi radiotransmitters and of the IEEE 802.11 MAC layer, so that the phenomenon of collisions is correctly represented. The discrete event simulator OMNeT++, and the associated INET framework, provide such a model that we will use. We have implemented both OLSR and DC for this experiment. A specific node is chosen to be the requester.

TABLE I: Simulation parameters

Area	500 x 500 m
Number of nodes	50, 100, 150, 200, 250
Average graph degree	5.16, 10.46, 15.69, 20.93, 26.16
Transmission range	100 m
Transmitter bitrate	10 Mbps
Applicative payload	24 B

D. Results

The first experiment evaluates the response rate of each algorithm. A requester sends a request, everyone is supposed to respond, and the responses are counted. We see on Fig. 6a that when MAC-level retries are disabled, DC achieves a much better reception rate than OLSR, regardless of variant. While OLSR has a reception rate of 30% on low-degree graphs and less than 10% on high-degree ones. At the same time, DC achieves between 55 and 70% for low-degree graphs and between 30% and 40% for high-degree ones. As expected, graphs with high degree are more susceptible to collisions, regardless of algorithm, even if DC resists much better. We also notice that overdimensioning the timeslots a bit seems to help the reception rate a bit: the balanced version of DC achieves a reception rate 10% to 15% higher than that of the optimistic version. Even if the L2 retries are disabled, dimensioning timeslots as if one of them is needed helps increase the reception rate slightly.

When the L2 retries are enabled (Fig. 6b), both OLSR and DC achieve near perfect reception rate, although OLSR tends to fall off at high-degree graphs. This does not mean that our algorithm is useless: in fact, in the case of OLSR, this near-perfect reception rate comes at a very high cost in transmitted bytes. Since OLSR generates many collisions, it has to make use of L2 retries very often, and this mechanism is a very expensive way of ensuring a good reception rate.

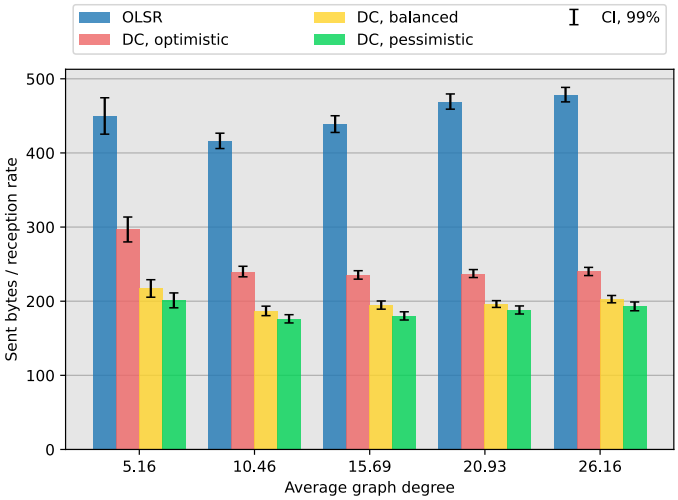


Fig. 7: Amount of bytes sent per received response (MAC-level retries activated)

We can see this clearly on Fig. 7 where the total amount of bytes sent for response messages is divided by the reception rate. We observe that DC is much more efficient than OLSR, needing less than half the bytes to achieve the same rate of received responses (for the balanced version). This comes from the fact that DC avoids collisions much better, and therefore doesn't need to use L2 retries as often as OLSR: DC combined with L2 retries is the most efficient way in terms of byte transmission to reach a perfect reception rate. We also see that overdimensioning the timeslots too much, as we did with the pessimistic version of DC, does not improve the efficiency significantly.

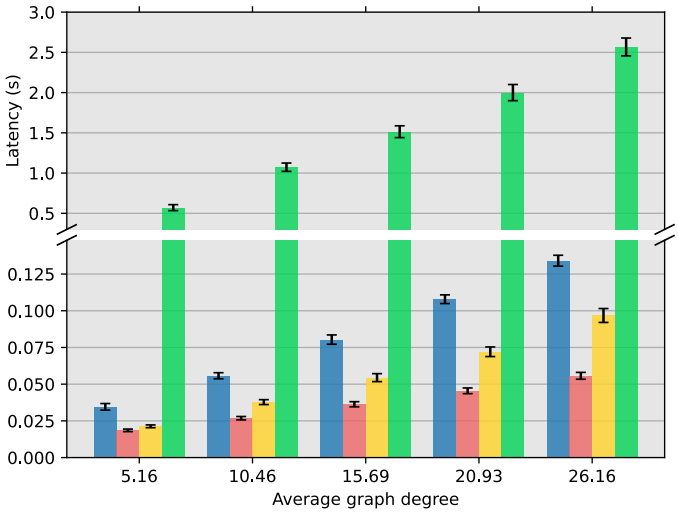


Fig. 8: Latency at 80% reception rate (MAC-level retries activated)

The most counter-intuitive result can be seen on Fig. 8. We define the latency as the time needed to achieve 80% reception rate, which was achieved for any configuration, as we've seen on Fig. 6b. We see that both optimistic and balanced versions of DC achieve lower latency than OLSR, despite the fact that DC adds deterministic delays on the responses. Optimistic DC achieves the same quorum of responses in less than half the time of OLSR regardless of graph degree, and even balanced DC achieves it in 50% of the time for low-degree graphs and 70% on high-degree ones. It turns out that delaying answers on purpose in order to avoid collisions is in fact better for latency than to risk those collisions and having to potentially retry the transmission: retries are costlier in terms of latency than deterministic delays.

We note that the latencies are low enough to make DC highly resistant to mobility. Our algorithm relies solely on information gathered during the broadcasting of the request, that is guaranteed to be very fresh. The probability that the graph changes between the broadcast and the convergecast phases is negligible, particularly if we have in mind a human pattern of mobility (people walking or cycling with handheld connected phones). In contrast, OLSR relies on the global information about the graph provided by the Topology Control message. Since these messages are costly[1], the RFC [3] recommends broadcasting them only every 5 seconds, and keeping the information provided by them up to 15 seconds.

We see that using the optional L2 retry mechanism provided by CSMA/CA is needed to achieve near-perfect response rate. Even if DC is much better at avoiding collisions than OLSR, it cannot (and does not attempt to) avoid all of them, due to the very limited amount of information it uses. As we can see, even doing a minimal effort of avoiding the very obvious collisions can help a lot decreasing the amount of bytes transmitted and the latency. DC does not aim to be a replacement to CSMA/CA-provided retries, but a complement to it.

V. Related works

Convergecast has mainly been studied in the context of Wireless Sensor Networks, where a fixed and static set of nodes must transmit some environment information to a base station regularly [6]. These nodes are typically small devices with low-bitrate and low-consumption antennae, very limited memory and a very little available power. The network is without infrastructure, and the base station is not directly connected with all the nodes, which means that the nodes must forward the packets from other nodes all the way to the base station. This paradigm is close to MANETs, with some important differences. Among those differences, we have a fixed set of nodes without mobility, a fixed base station that knows the entire link state of the graph and sometimes even the geographical position of the nodes, and synchronized clocks. These differences stem from the constraint of power saving inherent to WSN: it is essential to be able to switch off the radiotransmitter when idle. This constraint is not present in MANETs.

For this reason, WSN convergecast algorithms like [7, 8, 9] all try to build a spanning tree based on global graph information to schedule transmissions globally. In [9], the algorithm builds a breadth-first tree and allocates timeslots or channels to each node. The attribution of a unique parent to each node is done using the Euclidean distance criterion. In [8], the authors even assume that nodes are placed on a lattice, determine a period based on the collision distance of the nodes, and attribute initial timeslots according to the reception order of an initial broadcast: nodes choose their timeslot according to the node they receive the broadcast from. When this is done, each node can only transmit during the timeslots congruent to their initial timeslot modulo the period. In [7], the graph is reduced to a series of branches, and the information on each branch (number of nodes) is broadcasted to the entire network. The paper first offers a simple scheduling algorithm for converging packets from the whole branch to one of its ends, and then schedules the branches relatively to each other (each branch progressing in turn). More recently, [10] target low-duty-cycle WSNs, a

[1]Our simulations estimate the cost of TC messages to be respectively 430, 865, 1240, 1544 and 1763 bytes transmitted per node per second, according to the average degree of the graph.

specific type of WSNs where the nodes awaken at periodic time intervals. They assume that nodes know their neighbors, their level in the spanning tree rooted in the sink, as well as that of their neighbors and the active schedule of their neighbours. Starting backwards from the root, each node chooses a set of time-extended forwarders among the neighbors in the level above them, according to metrics (e.g. the expected probability of reception at the sink or the expected latency) that can be calculated recursively based on initial values fixed at the root. The time-extended forwarders assignation must be non-conflicting, which is always possible, but at the expense of latency (a node can always choose the same forwarder as one of its brothers, as long as it selects a later active timeslot). Although this paper is more interesting than the previous ones, since it does not assume that each node knows the entire structure of the graph and the forwarder choice is done in a decentralised manner, it still assumes synchronized clocks, and a fixed graph topology to amortize the high cost of initial calculations and messages.

A more interesting paradigm, closer to MANETs, is offered by [19]: only the base station is fixed, and broadcasts a beacon with a sequence number. Each retransmitter increases a hop counter, so that every node in the graph knows a shortest path to the base station. Next, the nodes send a beacon to the base station following this shortest path, so that the base station knows a spanning tree on the graph. A trivial convergecast algorithm can be built on this, but the collisions are left to CSMA/CA to deal with, and no aggregation mechanism is provided.

VI. Conclusion and future works

We have proposed a distributed convergecast algorithm that alleviates the issue of collisions when simultaneous responses from all the nodes of the network converge to the same node. Our algorithm, DC, is perfectly suited for generic MANETs that admit a limited (e.g. human) pattern of mobility, and is implementable on ubiquitous and application-independant hardware like connected phones equipped with WiFi antennae. It is taking into account the random behavior of CSMA/CA message delays, and uses its retry mechanism to its advantage when needed, without relying on it too much. DC can be used for a variety of applications that need to quickly converge information to a requester, but it was designed with round-based consensus algorithms like Paxos in mind, where a leader (who can be anyone and even change over time) needs a quorum to progress. DC will allow consensus algorithms to converge very fast, while also preserving the energy consumption of the nodes taking part in the consensus. We achieve this by using only local neighborhood information and a short-term spanning tree built during the broadcast phase, and locally scheduling responses that would otherwise collide.

Our experiments on a large sample of random connected dominating graphs show that we are able to be as effective as popular unicast algorithms like OLSR, while being more efficient and quicker by needing fewer transmission retries.

We have designed DC to be implementable on ubiquitous hardware communicating with the IEEE 802.11 WiFi protocol. It is also easily improvable to take into account recent improvements made to the WiFi protocol and implementations. We have assumed a single-channel and single-frequency WiFi for the sake of simplicity, but the algorithm is trivially adaptable to CDMA (Channel Division Multiple Access) and FDMA (Frequency Division Multiple Access) mechanisms provided in recent versions of the WiFi protocol. The local schedule we built can be improved to distribute those multiple channels and frequencies and further reduce collisions and latency.

References

[1] A. S. Tanenbaum and D. Wetherall, *Computer networks, 5th Edition*. Pearson, 2011.

[2] IEEE 802.11 Working Group, "Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-2016*, 2016.

[3] T. Clausen and P. Jacquet, "Optimized link state routing protocol (olsr)," *RFC*, vol. 3626, pp. 1–75, 2003.

[4] T. Clausen, P. Jacquet, C. Adjih, A. Laouiti, P. Minet, P. Mühlethaler, A. Qayyum, and L. Viennot, "Optimized link state routing protocol (olsr)," INRIA, Research Report 5145, 2003, 57 pp.

[5] P. Jacquet, P. Mühlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot, "Optimized link state routing protocol for ad hoc networks," in *IEEE Intern. Multitopic Conference*, 2001.

[6] S.-H. Yang, *Wireless Sensor Networks: Principles, Design and Applications*. Springer London, 2014.

[7] S. Gandham, Y. Zhang, and Q. Huang, "Distributed minimal time convergecast scheduling in wireless sensor networks," in *26th IEEE International Conference on Distributed Computing Systems*, 2006, p. 50.

[8] S. S. Kulkarni and U. Arumugam, "TDMA service for sensor networks," in *24th International Conference on Distributed Computing Systems Workshops*, 2004, pp. 604–609.

[9] V. Annamalai, S. K. S. Gupta, and L. Schwiebert, "On tree-based convergecasting in wireless sensor networks," in *2003 IEEE Wireless Communications and Networking*, 2003, pp. 1942–1947.

[10] L. Cheng, L. Kong, Y. Gu, J. Niu, T. Zhu, C. Liu, S. Mumtaz, and T. He, "Collision-free dynamic convergecast in low-duty-cycle wireless sensor networks," *IEEE Trans. Wirel. Commun.*, vol. 21, no. 3, pp. 1665–1680, 2022.

[11] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[12] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008.

[13] A. Qayyum, L. Viennot, and A. Laouiti, "Multipoint relaying for flooding broadcast messages in mobile wireless networks," in *IEEE annual Hawaii Intl. Conference on System Sciences*, 2002.

[14] D. Gutiérrez-Reina, S. L. Toral-Marín, P. Johnson, and F. J. Barrero-García, "A survey on probabilistic broadcast schemes for wireless ad hoc networks," *Ad Hoc Networks*, vol. 25, 2015.

[15] P. Ruiz and P. Bouvry, "Survey on broadcast algorithms for mobile ad hoc networks," *ACM Computing Surveys*, vol. 48, no. 1, 2015.

[16] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," *Wireless Networks*, vol. 8, no. 2-3, 2002.

[17] L. Viennot, "Complexity results on election of multipoint relays in wireless networks," INRIA, Research Report 3584, 1998, 16 pp.

[18] M. D. Penrose, *Random Geometric Graphs*. Oxford University Press, 2003.

[19] F. Araújo, J. Santos, and R. P. Rocha, "Implementation of a routing protocol for ad hoc networks in search and rescue robotics," in *IFIP Wireless Days, WD*, 2014, pp. 1–7.