Frugal Event Dissemination in a Mobile Environment*

Sébastien Baehni, Chirdeep Singh Chhabra, and Rachid Guerraoui

School of Computer and Communication Sciences, EPFL

Abstract. This paper describes an event dissemination algorithm that implements a topic-based publish/subscribe interaction abstraction in mobile ad-hoc networks (MANETs). Our algorithm is frugal in two senses. First, it reduces the total number of duplicates and parasite events received by the subscribers. Second, both the mobility of the publishers and the subscribers, as well as the validity periods of the events, are exploited to achieve a high level of dissemination reliability with a thrifty usage of the memory and bandwidth. Besides, our algorithm is inherently portable and does not assume any underlying routing protocol. We give simulation results of our algorithms in the two most popular mobility models: city section and random waypoint. We highlight interesting empirical lower bounds on the minimal validity period of any given event to ensure its reliable dissemination.

1 Introduction

The publish/subscribe (pub/sub) communication abstraction is a very appealing candidate for disseminating events in mobile ad-hoc networks (MANETs) [1]. In such networks, devices are mobile, they may not know each other and might not always be up and running. With a pub/sub abstraction, remote devices can communicate by playing two roles: the *publishers* produce events that are disseminated in the network and *subscribers* receive events they are interested in. Publishers and subscribers are decoupled in time, space and flow [2]. This makes the pub/sub abstraction appropriate for loosely coupled MANET applications.

Whereas the writing of MANET applications is appealing with a pub/sub abstraction, the effective implementation of such abstraction is not an easy task. In particular, ensuring a reasonable level of reliability of the dissemination is challenging without flooding the entire network. Indeed, devices in a MANET can directly broadcast information in their geographical neighborhood but need multiple indirections to reach far away devices. In addition, the devices typically run with a limited amount of memory and the dissemination algorithm cannot use a large portion of it just for buffering events. Similarly, the battery power of a device is (dynamically) limited and cannot anyway entirely be devoted to receiving and forwarding events, especially if those are duplicates or of no interest (i.e., *parasite* events).

* The work presented in this paper was sponsored both by the European IST PALCOM project (OFES No 03.0495-1), as well as by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

This paper presents an event dissemination algorithm that implements a topic-based pub/sub abstraction in MANETs. Our algorithm is inherently portable and does not assume any specific multicast routing protocol: we only rely on a standard Media Access Control (MAC) layer (e.g., Bluetooth [3] or 802.11 [4]). Events are (1) assumed to have a validity period that represents the time interval after which they are of no use and (2) are arranged according to a topic-hierarchy. The originality of our algorithm lies in its frugality, and this covers two aspects: first, despite the broadcast nature of the communication medium, we ensure that the subscribers receive a minimal number of duplicates and parasite events; second, both the mobility and validity periods of the events are used to enforce the reliability of the dissemination with a thrifty usage of the memory and bandwidth.

Our algorithm goes through three phases: (1) neighborhood detection based on exchanges of heartbeats in surrounding environments; (2) events dissemination after back-off periods calculated as functions of the frequency of the heartbeats and the number of events to send; and (3) garbage collection using the validity periods of the events as well as the number of times they have been propagated (a logical notion of "age").

We give simulations that highlight empirical lower bounds on the validity period needed to achieve a certain level of reliability. Interestingly, the lower bounds depend on the number of devices (publishers/subscribers), their speed, their interests (subscriptions), and the considered mobility models (i.e., random waypoint [5] or city section [6]). For instance, in the random waypoint model, an event with a validity period of 180 seconds is received by 95% of the 120 devices which move at 10 meters per second in an area of 25[km^2].

We compare our algorithm with three different flooding variants and show that, for the same reliability, our algorithm outperforms the alternatives in terms of bandwidth, duplicates and parasite events. For instance, for disseminating one event of 400 bytes in the very same previously described environment, we save between 300% and 450% of the bandwidth and each subscriber receives between 70 and 100 times less duplicates and between 50 and 90 times less parasite events.

The rest of the paper is structured as follows: Section 2 describes the MANET environment we consider. Section 3 gives an overview of the algorithm. Section 4 details the main elements of our algorithm. Section 5 gives various simulation results. Section 6 discusses related work and concludes our paper.[1]

2 Model

In this section we present some basic elements of the underlying MANET we consider. We discuss the communication medium, the network topology and the processes involved in the pub/sub interaction.

[1] An implementation of our algorithm for a parking application is given in [7]. The cars leaving the car parks act as publishers and propagate the information of free parking spots. When receiving such information, other cars, acting as subscribers, are able to locate the free place that is closest to their destination.

Overview. What we call a process in this paper is the piece of software of a mobile device that is responsible of disseminating/forwarding the events subscribed to by the application running on the device. We assume the processes to be mobile (they move with their host device) and to communicate directly with their immediate neighborhood (i.e., one-hop neighbors). A process can represent a publisher, a subscriber or both. All processes run our algorithm directly on top of the MAC layer (e.g., Bluetooth [3] or 802.11 [4]), without relying on any routing algorithm.

Communication Medium. The range of a process is the geographical zone within which it can directly reach other processes using a simple send communication primitive of the underlying MAC layer (one-hop). The set of processes in the range of a process p_i is called the neighborhood of p_i. A process cannot send a message to only one of its neighboring processes nor directly send a message to processes multiple hops away (i.e., no underlying unicast/multicast routing algorithm is assumed).

Network Topology. We assume that the network is completely ad-hoc and no fixed infrastructure is present. We do not make any assumption on the size of the network (number of processes), nor on the connection graph of the processes. In particular, the graph does not need to be fully connected at any given point in time. The processes are assumed to be mobile. When analyzing our algorithm, we will study the two most popular mobility models: (1) random waypoint [5] and (2) city section [6], which we recall below.

- In the random waypoint model, a process moves from its current location to a new location by randomly choosing a direction and a speed. The speed and direction are chosen from pre-defined ranges, [*speedmin, speedmax*] and $[0, 2\pi]$ respectively. This model includes pause times between changes in direction and/or speed.
- In the city section model, the mobility area is a street network that typically represents a section of a city. In this model, the processes follow predefined guidelines like speed limits, one way lanes, and other traffic laws. Each process begins the simulation at a predefined point on some street, and randomly chooses a destination. It is common to consider specific characteristics like pause times, acceleration and deceleration in certain intersections.

Processes, Topics and Events. Each process p_i has a unique identifier i. All processes have to deal with limited bandwidth, energy and memory. A process can move in and out of the range of other processes, or crash (or recover), at any time.

Each event $e_j^{T_k}$ published by a process p_i: (1) has a unique identifier j,[2] (2) a validity period, i.e., $val(e_j^{T_k}) = t$, after which the information carried by the event is of no use in the system, and (3) is associated to a specific topic, e.g., T_k. Topics are arranged in a hierarchy (e.g., *.grenoble.conferences.middleware*) and a subscriber that subscribes to a specific topic (e.g., *.grenoble.conferences*) is expected to receive events of this topic and all its subtopics (e.g., *.grenoble.conferences.middleware*). The root topic of the topic tree is denoted by the *dot (.)* sign. An event of a topic, which a process has not subscribed to, is called a *parasite* event for that process.

[2] In the paper, we assume, without loss of generality, that the size of the event identifier is smaller than the size of the data carried by the event.

3 Algorithm Overview

We give here an overview of our algorithm before detailing it in subsequent sections. Our algorithm goes through three phases: (1) neighborhood detection, (2) event dissemination and (3) garbage collection. We first introduce these phases and then give a short example to illustrate their execution.

Phase 1: Neighborhood detection. The processes periodically exchange heartbeat messages, each contains the following elements: (1) the identifier of the process, (2) a list of its current subscriptions (i.e., a list of topics T_i, T_j, ..., T_n)[3] and, (3) its current speed (this information is only useful for optimization purpose and is not mandatory). Each process p_i uses the heartbeat messages it receives to construct a dynamic one-hop neighborhood table, containing the identifiers of the processes in the neighborhood along with their subscriptions and their current speed (if available). Only the processes whose subscriptions match with the ones of p_i, are kept in p_i's table. Other one-hop neighbors are of *no interest* to p_i. The neighborhood table is continuously garbage collected and updated (depending on the periodicity of the heartbeats). If the speed information of the processes is available (for example with the help of a tachometer), the process can adjust the periodicity of the heartbeats to match to the dynamicity of its environment. Otherwise, this periodicity is set to a static value (see Section 4.2).

When processes detect each other, they exchange a list of identifiers of the events they have kept after receiving them and which are still valid. When receiving event identifiers, each process checks if its neighbor is interested in an event it has not already received (i.e., needs the event). In this case, the processes proceed to the dissemination phase (see below). Sending the events identifiers instead of the events themselves preserves network bandwidth and CPU processing power. Indeed, it might happen that a process p_i has already received the same events as process p_j. Consequently, it makes no sense for p_j to send these events to p_i again.

Phase 2: Dissemination. When a process detects that one of its neighbor needs an event (when comparing the list of events identifiers it receives with its own list of events), it broadcasts the required event to its neighborhood together with the list of its interested neighbors, after a back-off period (see Section 4.2).

After receiving the event, the neighboring processes of the sender might decide to propagate the event if they know other processes, in their neighborhood, that have not yet received it and that are interested in it (see Section 4.3). If the processes that receive the event have subscribed to the topic of this event and have not received it yet, they deliver it to the application and store it, until it is garbage collected. A process that receives an event it is not interested in (parasite event), simply drops it. This way, we minimize the burden induced by parasite events and save valuable memory.

Phase 3: Garbage collection. Throughout the two previous phases of our algorithm, we mainly use two main data structures (see Section 4.1) at every process.[4] The first

[3] This list can change at any point in time with respect to the interests of the process.
[4] Other data structures are involved in the algorithm, but those cannot induce memory problems.

one is used for storing the list of neighbors that shares the same subscriptions as the process itself (neighborhood table). The second one is used for storing the events. The neighborhood table is constantly updated (based on the periodicity of the heartbeats) and its size is bounded.[5]

The data structure used to store the events can grow rapidly. This is because the total number of events published in the system is unbounded and the processes have to store them until their validity period expires. It can thus happen that a process receives an event and cannot store it because its memory is full. Our garbage collection scheme collects, every time a new event has to be stored and if the memory is full, the events according to their validity period and the number of times they have been propagated (sent/forwarded) by the processes.

Illustration. Figure 1 depicts a simple scenario illustrating the three phases of our algorithm. We consider a hierarchy made of three topics: T_0, T_1 and T_2; T_1 is a subtopic of T_0 whereas T_2 is a subtopic of T_1. Three processes, p_1, p_2 and p_3 are involved: p_1 has subscribed to T_1, p_2 has subscribed to T_2 and p_3 has subscribed to T_0. Three events are published in the system: $e_3^{T_1}$, $e_4^{T_2}$ and $e_5^{T_2}$. We assume that p_1 has already received $e_3^{T_1}$ and p_2 has already received $e_4^{T_2}$ and $e_5^{T_2}$.

In part I of Figure 1, processes p_1 and p_2 become neighbors and hence know their common subscriptions. They then exchange the identifiers of the events corresponding to the topics they have commonly subscribed to. As a consequence, p_2 sends to p_1 events $e_4^{T_2}$ and $e_5^{T_2}$ (as T_1 is a super-topic of T_2).

In part II of Figure 1, all three processes become neighbors, and exchange their event identifiers: p_1 and p_2 realize that p_3 misses events, $e_3^{T_1}$, $e_4^{T_2}$ and $e_5^{T_2}$. As both p_1 and p_2 have events to send, they both send them after a back-off period. It is important to notice that, because p_1 has more events to send than p_2, p_1 has a smaller back-off period than p_2 (see Section 4.3).

In part III of Figure 1, p_1 moves on, but p_2 and p_3 still remain in range. As p_2 was in the range of p_1 when it sent the events list, p_2 heard the events that p_1 sent for p_3. Now, p_2 and p_3 know that they do not have to exchange events anymore.

4 Algorithm Description

In this section we first detail the data structures involved in the algorithm. Then we describe the neighborhood detection, the dissemination and finally the garbage collection.

4.1 Data Structures

As illustrated in Figure 4, we consider a list of *subscriptions* for every process p_i (p_i*.subscriptions*), a *neighborhood table* (*neighborhoodTable*) and an *event table* (*eventsTable*). These two tables are detailed below. There is also the list containing the *events to send* (*eventsToSend*). The different parameters used, as listed in Figure 4, are: the heartbeat delay (*HBDelay*), the neighborhood garbage collection delay (*NGCDelay*) and the back-off delay (*BODelay*).

[5] The upper bound corresponds to the maximum number of neighbors a process can handle. This bound depends on the structure of the network and on the amount of memory of the processes.

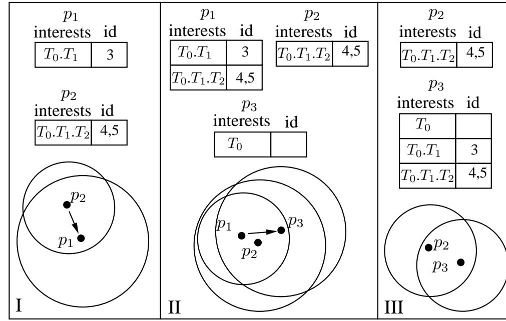

Fig. 1. Illustration of our algorithm

Subscriptions of a process. The different subscriptions of every process p_i are stored in the list p_i.*subscriptions*. We assume, without loss of generality, that the size of this list is bounded as the number of subscriptions of a process is usually limited in the topic-based scheme. In this scheme, a process only has to subscribe to a topic to receive all the events regarding this topic and all its subtopics. A process can change the list of its subscriptions at any time.

Neighborhood Table. Figure 2 illustrates the neighborhood table of a process. The first column of this table stores the identifiers of the neighbors of a process. The second column stores the topics those processes have subscribed to. The third column stores the identifiers of the events the neighbors have received, the fourth column contains the speed of the neighbors (this column is not mandatory and the speed of the processes is only used for optimization purpose) and the last column contains the time when the entry has been stored/updated into the table. This last entry is used for garbage collection purpose. We discuss in more detail the use of the neighborhood table in Section 4.3 and present its garbage collection algorithm in Section 4.4.

Neighbors	Topics	Events ID	Speed	Store Time
1	T_0	1, 2	1 [mps]	07:45:23
32	$T_0.T_1.T_2$	10	– [mps]	07:43:20
542	$T_0.T_4$	210	20 [mps]	07:44:45

Fig. 2. Neighborhood table

Topic Hierarchy

T_0 → T_1, T_4; T_1 → T_2, T_3

Topics	Events	Validity	Counter
$T_0.T_1.T_2$	$e_{10}^{T_2}$	100 [s]	5
	$e_5^{T_2}$	60 [s]	1
$T_0.T_1.T_3$	$e_3^{T_3}$	20 [s]	2
	$e_{143}^{T_3}$	120 [s]	12

Fig. 3. Event table

Event Table. Each process stores an event table as shown in Figure 3. This table contains a list of topics the process has subscribed to, together with the list of events this process has received and/or published. These events are stored according to the topic

hierarchy (from the partial topic tree information the process has). Each event has a unique identifier (*id*), a validity period (*validity*), a counter (*counter*), a topic (*topic*) and its internal data information (*data*: this information is not shown in Figure 3). The validity period expresses the time interval after which the event can be removed from the system. The counter represents the number of times an event has been forwarded; it is used, together with the validity period, in the garbage collection phase (see Section 4.4).

The events to send. This structure contains the events a process sends to its neighbors. This structure can be, at most, as big as the event table (if a process has to send all its events to its neighbors). The structure is reset each time the events are sent (i.e., after each back-off).

4.2 Neighborhood Detection

Before detecting neighbors, the processes have to subscribe to topics they are interested in. The subscription/unsubscription sub-protocol is depicted in Figure 5. Basically, when a process wants to subscribe to a specific topic, it adds this topic to its list of subscriptions and starts the *heartbeat* and *neighborhoodGC*[6] tasks. A process that wants to unsubscribe to a topic, removes this topic from its list of subscriptions. When the list of subscriptions is empty, the *heartbeat* and *neighborhoodGC* tasks are stopped.

For each process p_i

```
1: {The subscriptions of the process}
2: p_i.subscriptions = ∅;
3: {The neighborhood table }
4: neighborhoodTable = ∅;
5: {The event table}
6: eventsTable = ∅;
7: {The structure containing the events to send}
8: eventsToSend = ∅;
9: {The default heartbeat delay}
10: HBDelay = 15000;
11: {The default neighborhood garbage collection delay}
12: NGCDelay = HBDelay*HB2NGC;
13: {The default back-off delay}
14: BODelay = HBDelay / HB2BO;
```

Fig. 4. Data structures

For each process p_i

```
1: {The subscription algorithm}
2: upon SUBSCRIBE(T_k) do
3:    p_i.subscriptions = p_i.subscriptions ∪ T_k;
4:    if (HEARTBEAT not started) then
5:       start HEARTBEAT;
6:    end if
7:    if (NEIGHBORHOODGC not started) then
8:       start NEIGHBORHOODGC;
9:    end if
10: end upon

11: {The unsubscription algorithm}
12: upon UNSUBSCRIBE(T_k) do
13:    p_i.subscriptions = p_i.subscriptions \ T_k;
14:    if (p_i.subscriptions == ∅) then
15:       stop HEARTBEAT; stop NEIGHBORHOODGC;
16:    end if
17: end upon
```

Fig. 5. Subscription, unsubscription

The heartbeats of a process carry the list of subscriptions of the process (e.g., "T_0, T_1,..., T_n") along with its process identifier and its current speed. As we pointed, the

[6] The *neighborhoodGC* task is used for garbage collecting the entries of the neighbors' identities from the neighborhood table; it is presented in Section 4.4.

information about the speed of the processes is not mandatory and is only used as an optimization. For instance, this information can be used to tune the number of heartbeat messages according to the speed of the process and the speed of its neighbors. In a dynamic environment, the delay between two heartbeats could be set to a shorter period than in a more static one.

After receiving the heartbeat messages, each process builds a view of its neighborhood, together with a list of their subscriptions. If two neighboring processes do not share any common topics, these topics are not stored in their respective neighborhood table. The neighborhood information of a process is stored in a specific table (Figure 2) and updated accordingly (using the UPDATENEIGHBORINFO() method[7]).

```
For each process p_i

1:  {The heartbeat task}
2:  task HEARTBEAT
3:     SEND(i,p_i.subscriptions, [currentSpeed]);
4:  end

5:  {When receiving a heartbeat message}
6:  upon RECEIVE(j,subscriptions,[speed]) do
7:     if subscriptions ∈ p_i.subscriptions then
8:        RAISE new neighborEvent(j,subscriptions);
9:        if (j ∉ neighborhoodTable) then
10:          neighborhoodTable ∪
               (j,subscriptions,[speed],currentTime);
11:       else
12:          UPDATENEIGHBORINFO(j,
               subscriptions,[speed],currentTime);
13:       end if
14:    end if
15:    COMPUTEHBDELAY(neighborhoodTable);
16:    COMPUTENGCDELAY();
17: end upon

For each process p_i

18: {A new neighbor has been detected}
19: upon new neighborEvent(j,subscriptions) do
20:    if subscriptions ∈ p_i.subscriptions then
21:       SEND(i,GETEVENTSIDS(subscriptions,
             eventsTable));
22:    end if
23: end upon

24: {Reception of a list of events identifiers}
25: upon RECEIVE(j, eventsIDs) do
26:    if j ∈ neighborhoodTable then
27:       for all eventID ∈ eventsIDs do
28:          UPDATENEIGHBOREVENTINFO(j, eventID,
               currentTime);
29:       end for
30:       RETRIEVEEVENTSTOSEND();
31:    end if
32: end upon
```

Fig. 6. Neighborhood detection

If the subscriptions of a process match the ones of its neighbor, they then exchange the event identifiers they have subscribed to (the event identifiers are retrieved via the GETEVENTSIDS() method[8]). Once those event identifiers are received, the process updates its neighborhood table with those and checks if it has to send events to its neighbors (via the RETRIEVEEVENTSTOSEND() method, described in Section 4.3). The identifiers of the events are exchanged instead of the actual events to minimize the duplicate messages. It may happen that a process and its neighbors have the same set of events; in this case, there is no need for them to exchange the events.

The computation of the time intervals for (1) the heartbeat messages, (2) the neighborhood garbage collection and (3) the back-off period are determined at the reception

[7] This method is omitted for space limitations. It simply consists of updating the information (i.e., subscriptions, speed and store time) corresponding to the right neighbor.

[8] Again, this method is omitted for space limitations. It consists in retrieving, from the *eventsTable*, the event identifiers of the received events corresponding to a certain topic.

```
For each process p_i

1: {Computation of the events to send}
2: function RETRIEVEEVENTSTOSEND()
3:   eventsToSend = ∅;
4:   for all neighbor ∈ neighborhoodTable do
5:     if neighbor.subscriptions ∈ p_i.subscriptions then
6:       for all e_k^{T_j} ∈ eventsTable do
7:         if T_j ∈ neighbor.subscriptions &&
           k ∉ neighbor.eventsIDs &&
           val(e_k^{T_j}) < currentTime then
8:           eventsToSend ∪ e_k^{T_j};
9:         end if
10:      end for
11:    end if
12:    if eventsToSend ≠ ∅ then
13:      COMPUTEBODELAY();
14:      if backOff not started &&
           BODelay != null then
15:        start backOff with computed BODelay;
16:      end if
17:    end if
18:  end for
19: end
```

Fig. 7. Event retrieval

```
For each process p_i

1: {Computation of the hearbeat delay}
2: function COMPUTEHBDELAY(neighborhoodTable)
3:   averageSpeed =
     AVERAGESPEED(neighborhoodTable);
4:   if averageSpeed ≠ null then
5:     HBDelay = x / averageSpeed;
6:   end if
7:   HBDelay = MIN(HBDelay, heartbeat upper bound);
8:   HBDelay = MAX(HBDelay, heartbeat lower bound);
9: end

10: {Computation of the neighborhood
    garbage collection delay}
11: function COMPUTENGCDELAY()
12:   NGCDelay = HBDelay*HB2NGC;
13: end

14: {Computation of the back-off delay}
15: function COMPUTEBODELAY()
16:   if BODelay == null then
17:     BODelay =
        HBDelay / (HB2BO*sizeof(eventsToSend));
18:   else
19:     BODelay = MIN(BODelay,
        HBDelay / (HB2BO*sizeof(eventsToSend)));
20:   end if
21: end
```

Fig. 8. Computing delays

of the heartbeat messages, using respectively the following methods: (1) COMPUTEHBDELAY(), (2) COMPUTENGCDELAY() and (3) COMPUTEBODELAY(). Figure 8 describes an implementation of these methods. Parameter x represents a variable the programmer can use to tune the heartbeat delay with respect to the average speed of the processes (for instance, x can represent the propagation radius of the wireless device). Parameters *HB2BO*, respectively *HB2NGC*, represent the factors by which the heartbeat delay is divided, respectively multiplied, in order to set the periodicity of the back-off delay, respectively the neighborhood garbage collection delay.

4.3 Dissemination

Our dissemination scheme algorithm is described in Figure 9. Basically the process uses the PUBLISH() method to send the event to the neighboring processes if at least one of those has subscribed to the topic of the event. In calling this method, the process updates the neighbor information in its neighborhood table (via the UPDATENEIGHBOREVENTINFO() method[9]).

As soon as a process receives an event, it updates its neighborhood table (using the UPDATENEIGHBOREVENTINFO() method) with the list of neighbor identifiers it

[9] For space limitation, this method is not shown in the algorithm; it basically consists in updating the list of the presumed received events of a neighbor with the event identifier given as a parameter.

received with the events. The process then checks if it has subscribed to the topic of that event and if so, it delivers it to the application and adds it to its event table (after checking that the event table is not full, otherwise it calls the GARBAGECOLLECT() method). If the process has not subscribed to the topic of the event, it simply drops it. Once it has delivered the event to the application, the process checks if it has to forward its events to its neighbors (i.e., RETRIEVEEVENTSTOSEND() method, Figure 7).

If a process p_i finds out that some of its neighbors have subscribed to the topic of the still valid events p_i owns, p_i starts a back-off period (the back-off delay is determined by the function COMPUTEBODELAY()[10]). Taking into account the events that have been received by the processes reduces the number of useless retransmissions and hence prevents duplicates and saves bandwidth.

Once the back-off delay expires, the events to send are recomputed (in case the neighborhood of the process has changed between the beginning and the end of the back-off or if the validity period of an event expires) and the new events are sent, together with a list of its neighbor identifiers. The sending process then updates its neighborhood table and increments the counter of each event that has just been sent.

4.4 Garbage Collection

We present here how the different data structures are garbage collected in order to conserve the sparse memory optimally.

Subscription list of a process. As stated in Section 4.1, we can assume that the size of this data structure is limited and the information it contains is constantly updated when the process decides to subscribe or unsubscribe to specific topics.

Neighborhood table. Each time the *neighborhood garbage collection delay* expires, the process identities whose store times have expired are collected from the neighborhood table (see Figure 10). As this task is executed periodically and as we assume that the total number of neighbors is limited, the size of the table is bounded.

Event table. Each time a new event has to be stored in the *eventsTable*, a check to test if the memory is full is done. If the check succeeds, one event, whose validity period has expired, is garbage collected. If all the events in the *eventsTable* are still valid, we run a garbage collection algorithm based on the notion of validity period and on the number of times an event was propagated. This algorithm ensures that events with high validity periods that have been propagated several times are garbage collected before events with short validity periods that have never been forwarded. Equation 1 captures the way we collect the events, based on: (1) their validity period (i.e., val($e_k^{T_j}$)) and (2) the number of times an event has been forwarded (i.e., fwd($e_k^{T_j}$)). The garbage collection function for an event $e_k^{T_j}$ is given as ($\forall val(e_k^{T_j}), fwd(e_k^{T_j}) \in \mathbb{N}^*$):

[10] An implementation is given in Figure 8. In this implementation, the back-off delay depends on the heartbeat delay and the total number of events to send.

For each process p_i

```
1: {Executed when the back-off expires}
2: upon backOff expiration do
3:    BODelay = null;
4:    if eventsToSend ≠ ∅ then
5:       SEND(i, eventsToSend, neighborsIDs);
6:       eventsIDs = GETEVENTSIDS(eventsToSend);
7:       for all neighborID ∈ neighborhoodTable do
8:          for all id ∈ eventsIDs do
9:             UPDATENEIGHBOREVENTINFO(neighborID,
               id);
10:         end for
11:      end for
12:      INCREMENT(eventsToSend, eventsTable);
13:   end if
14: end upon

15: {Reception of a list of events}
16: upon RECEIVE(j, events, neighborsIDs) do
17:    for all e_k^{T_j} ∈ events do
18:       for all neighborID ∈ neighborsIDs do
19:          UPDATENEIGHBOREVENTINFO(neighborID,
             k);
20:       end for
21:       if T_j ∈ p_i.subscriptions &&
          e_k^{T_j} ∉ eventsTable then
22:          interested = true; STOP backOff timer;
23:          if eventsTable is full then
24:             garbageCollect(eventsTable);
25:          end if
26:          eventsTable ∪ e_k^{T_j} ;DELIVER(e_k^{T_j});
27:       end if
28:    end for
29:    if interested then
30:       RETRIEVEEVENTSTOSEND();
31:    end if
32: end upon
```

For each process p_i

```
33: {Publication of a new event e_k^{T_j}}
34: function PUBLISH(i, e_k^{T_j}, neighborsIDs)
35:    for all neighbor ∈ neighborhoodTable do
36:       if neighbor.subscriptions ∈
          p_i.subscriptions then
37:          interested = true; break;
38:       end if
39:    end for
40:    if interested then
41:       SEND(i, e_k^{T_j}, neighborsIDs);
42:       for all neighborID ∈ neighborhoodTable do
43:          UPDATENEIGHBOREVENTINFO(neighborID,
             k);
44:       end for
45:    end if
46:    if eventsTable is full then
47:       garbageCollect(eventsTable);
48:    end if
49:    eventsTable ∪ e_k^{T_j}; DELIVER(e_k^{T_j});
50:    if (NEIGHBORHOODGC not started) then
51:       start NEIGHBORHOODGC;
52:    end if
53: end
```

Fig. 9. Dissemination

For each process p_i

```
1: {Garbage collection of the neighborhood
   table}
2: task neighborhoodGC
3:    for all neighbor ∈ neighborhoodTable do
4:       if currentTime - NGCDelay > neighbor.storeTime
         then
5:          REMOVE(neighbor,neighborhoodTable);
6:       end if
7:    end for
8: end
```

For each process p_i

```
9: {Garbage collection of the events table}
10: function garbageCollect(eventsTable)
11:    gc = null;
12:    for all e_k^{T_j} ∈ eventsTable do
13:       if val(e_k^{T_j}) > currentime then
14:          gc = e_k^{T_j}; break;
15:       end if
16:       if val(e_k^{T_j}) / (fwd(e_k^{T_j}) + val(e_k^{T_j})) ≤
          val(gc) / (fwd(gc) + val(gc)) then
17:          gc = e_k^{T_j};
18:       end if
19:    end for
20:    REMOVE(gc,eventsTable);
21: end
```

Fig. 10. Garbage collection

$$gc(e_k^{T_j}) = \frac{val(e_k^{T_j})}{(fwd(e_k^{T_j}) + val(e_k^{T_j}))} \quad (1)$$

For instance, an event with a validity period of 2[min] that has been forwarded less than 2 times, will be collected after an event with a validity period of 5[min] that has been forwarded 5 times.

The events to send. As discussed in Section 4.1, the data structure capturing the events to be sent does not need to be garbage collected as it is reset every back-off period. Moreover, its size depends on the size of the event table, but as this data structure is efficiently garbage collected, the size of the *events to send* list cannot grow indefinitely.

5 Performance

We present here performance results obtained from simulating our algorithm, according to the two popular mobility models. We first describe the simulation setting and then give the actual performance measurements.

5.1 Environment

Our algorithm was simulated using *Qualnet 3.7* [8], directly on the 802.11b MAC layer, in the two different mobility models: (1) random waypoint [5] and (2) city section [6].

Configuration Parameters. The size of the events is set to 400 bytes, *x* to 40, *HB2BO* to 2 and *HB2NGC* to 2.5. The heartbeat upper bound period is set to 1[s] for the random waypoint model and varies in the city section model. The mobility of the processes and the validity periods of the events vary (see the following performance measurement configuration). The choice of these values (i.e., *x*, *HB2BO* and *HB2NGC*) reflects a trade off between the overall number of messages sent (heartbeats, events identifiers, and actual events) and the reliability of the dissemination. For the random waypoint model, the data were gathered after the first 600 seconds of the simulation time (due to the high variability in the neighborhood percentage during these first seconds [9]).

Random Waypoint in Qualnet. In our experiments, the pause time is always set to 1[s]. The maximum and minimum speed vary during the entire set of experiments, see Section 5.2. Moreover, in this model, we have conducted our experiments on a virtual area of 25[km^2], populated randomly with 150 processes. Regarding the overall settings of the simulator, a "standard" 802.11b ad-hoc network was used. The transmission power is 15[db] for all the rates 1,2,6 and 11[Mbps], whereas the reception sensitivity is -93[db], -89[db], -87[db] and -83[db] for 1,2,6 and 11[Mbps] respectively.[11] The channel frequency is 2.4[Ghz] and uses a statistical propagation model, with a limit of -111[dbm] and a two ray path loss model. Each process has an omni-directional antenna with an efficiency of 0.8.

[11] This corresponds to a radio range of a sphere which radius is 442[m], 339[m], 321[m] and 273[m] respectively.

City Section in Qualnet. For this model, the map of our campus at EPFL was chosen and a specific mobility model for 15 processes was created. The EPFL campus covers 1200x900[m^2]. The processes do not walk/drive randomly on each of the roads. The real traffic conditions were considered (e.g., some roads are more often used than others). The overall settings of the simulator are the same as for the random waypoint, except for the reception sensitivity which is -65[db] for all rates (1,2,6 and 11[Mbps])[12]. We have adapted these values to simulate the real radio range of a city.

5.2 Results

Random Waypoint Model. We conducted the simulation for different speeds: 0[mps], 1[mps], 5[mps], 10[mps], 20[mps], 30[mps] and 40[mps]. All the simulations were run 30 times with different initialization (i.e., seed) values and the results presented in each case were averaged over the 30 obtained values. One event is published in each case.

In the first experiment, the validity period of the events and the speed of the processes were considered. The plain and dashed graphs of Figure 11 represent reliability values obtained when only 20% and 80%, of the processes, have respectively subscribed to the topic of the event. We can see that, when few processes have subscribed to that topic (20%), it is very difficult to achieve high reliability, unless if the processes move at high speed. We can explain this by the fact that the area is far too big with respect to the number of subscribers. If only 20% of them have subscribed to the topic of the event, we end up with only 30 processes for a region of 25[km^2]; the network is too sparse. However, when more processes have subscribed to the topic (80%), we can achieve a fairly high reliability with different validity periods and different speeds of the processes. For example, processes moving at 10[mps] and publishing events with a validity period of 180[s] have the same 95% reliability than processes moving at 30[mps] and publishing events with a validity period of 90[s]. Interestingly, under some lower bounds of validity period, it is possible, to achieve a specific reliability given different mobility models and speeds of the processes.

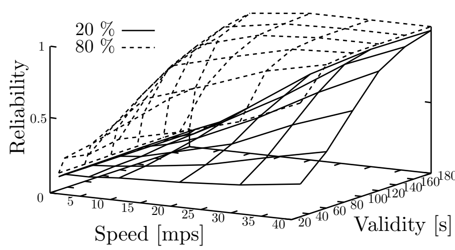

Fig. 11. Probability of event reception as a function of the validity period, the speed of the processes and the number of subscribers

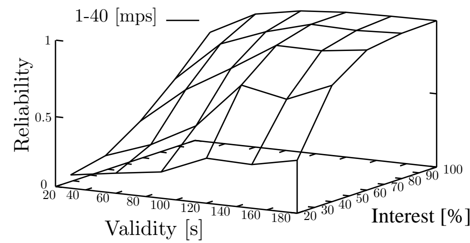

Fig. 12. Probability of event reception as a function of the validity period and the number of subscribers, in a heterogeneous mobile environment

[12] This corresponds to a radio range of a sphere which radius is 44[m].

In Figure 12, we depict the same experiments as before, except that now we have a more heterogeneous mobile network, in which the processes randomly move between 1[mps] and 40[mps]. With a low number of subscribers, the reliability is low also. However, even if only 60% of the processes have subscribed to the topic of an event with a validity period of 120[s], all of them receive the event. We can relate these results to the ones of a network in which all processes move at a speed of 20[mps]. Indeed, according to our results, the overall reliability depends on the validity period and the average speed of the processes in the network, rather than on the specific speed of each process.

City Section Model. In this model, all 15 processes drive at a given speed which is the speed limit of the road they are currently driving on (which is between 8[mps] and 13[mps]) and it may happen that they stop for a while for several reasons (red light, parking etc.). In all experiments, all processes, in turn, become the original publisher. This basically means that the original publisher is not always the same process but changes for each experiment. Again, all experiments were conducted 30 times and the results we present are an average over these 30 times for the 15 publishers.

In the first set of experiments, the importance of the heartbeat period over the overall reliability was measured. In such a network, with no upper bound set, the processes send heartbeats every 4[s] (which is the fraction of x over the average speed of 10[mps]). Figure 13 depicts the different results obtained when varying the heartbeat upper bound period from 1[s] to 5[s], where all the processes have subscribed to the topic of the event and where the validity period of this event is 150[s].

Heartbeat upper bound period [s]				
1	2	3	4	5
76.9%	75.1%	65.5%	69.9%	54.0%

Fig. 13. Probability of event reception as a function of the heartbeat period

Subscribers [%]				
20%	40%	60%	80%	100%
58.1%	59.7%	62.5%	68.6%	76.9%

Fig. 14. Probability of event reception as a function of the number of subscribers

We can notice that there is no real difference in reliability between the heartbeats sent every 1[s] or 2[s]. However, between 1[s]-2[s] and 5[s], we have a loss of 22% reliability. Interestingly, having heartbeats every 4[s] is better than having them every 3[s]. This surprising result is explained by the fact that, with this heartbeat period of 3[s], the messages sent by the processes are more likely to collide.

In the second set of experiments, the heartbeat upper bound period was set to 1[s] and the number of subscribers varied from 20% to 100%. Interestingly, these results are not comparable with the ones obtained in the random waypoint model. Indeed, even if only 20% of the processes have subscribed to the topic of the event, almost 60% of them receive the event which is better than the previous model. This can be explained by the fact that, in this model, the processes follow specific paths defined according to specific rules, so they are more likely to become neighbors than in the random waypoint model, especially if certain roads have more importance than others (which was the case in our

simulations). We also point out the importance of the path taken by the processes when we compare the reliability achieved by each of the publishers. In Figure 15, we depict the maximum difference between the minimum reliability and the maximum reliability between the publishers, for different percentage of subscribers. There can be a huge difference of reliability between the publishers that originally publish the event and this difference is due to the path taken by the publisher.

Subscribers [%]				
20%	40%	60%	80%	100%
40.9%	44.7%	47.9%	53.9%	60.0%

Fig. 15. Difference of reliability between the processes

Event Validity Period[s]					
25	50	75	100	125	150
11%	27%	44%	52%	69%	77%

Fig. 16. Probability of event reception as a function of the event validity period

In the third set of experiments, the heartbeat upper bound period was set to 1[s] and the validity period of the events varied between 20[s] and 150[s]. In Figure 16, we can see that the validity period of the event has a crucial importance on the overall reliability. This comes from the fact that, in this specific model, we cannot distinguish *where* and *when* the processes become neighbors. In the random waypoint model, the processes exchange information uniformly during the simulation: there is no real hot-spot where the processes meet. On the contrary, in the city section model, the processes are more likely to meet and exchange their information at social meeting points, hence the huge differences in reliability.

Frugality. To quantify the frugality of our algorithm, it was compared with three alternative approaches: (1) simple flooding, (2) simple flooding while taking into account the interests of the subscriber (interests-aware flooding) and (3) simple flooding in taking into account the interests of both the subscriber and its neighbors (neighbors' interests flooding). In (1), an event is sent every second by a process to all its neighbors which in turn, irrespective of their interests, propagates it with the same technique. In (2), the processes, at every one second interval, propagate only the events they are interested in. In (3), a process propagates an event to its neighbors only if the process itself and its neighbors are interested in the event. We compared four different measurements: (1) the bandwidth used per process, (2) the number of events sent per process, (3) the number of duplicates received per process and (4) the number of parasite events received per process.

All of the following measurements were averaged over 30 experiments and have been done using the random waypoint model described above with the speed of the processes set to 10[mps] (in order to compare the approaches with the same reliability degree)[13]. The size of one heartbeat was set to 50 bytes and the size of one event identifier to 128 bits. We varied the number of subscribers from 20% to 100% as well as the number of events from 1 to 20 (the size of one event has been set to 400 bytes).

[13] Please note that approach (1) as always 100% reliability, due to its inherent behavior.

Figure 17 shows the bandwidth used per process during a simulation of 180[s].[14] Our algorithm consumes less bandwidth than the other approaches in every cases, except if the sum of the events' sizes is lower than 1,5[kB] and the number of interested processes is less or equal to 20%. In this very special case, the second alternative is better. However, our algorithm is much less sensitive to the size of the events as we send very few of them (see Figure 18). Our algorithm sends between 50 to 100 times lesser events compared to the other alternative approaches. Consequently, if one event is of size 1.6[kB] instead of 400 bytes, we outperform every other alternatives, even for a small number of events published and a small number of subscribers.

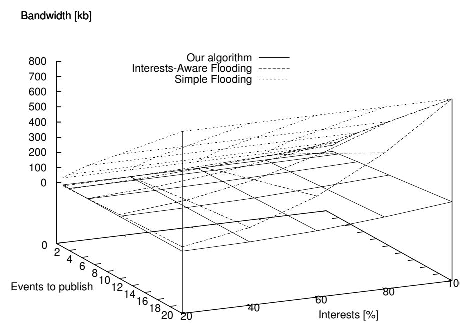

Fig. 17. Bandwidth usage per process as a function of the number of events to publish and the number of subscribers

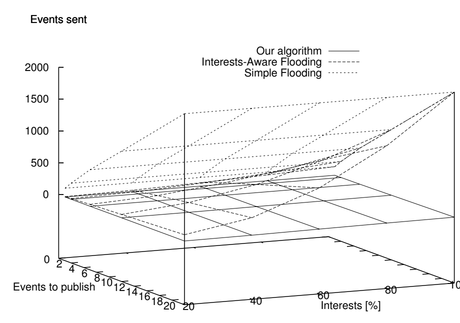

Fig. 18. Number of events sent per process as a function of the number of events to publish and the number of subscribers

Figure 19 depicts the number of duplicates received per process during the 180[s].[15] Our algorithm outperforms approach (2) by a factor varying from 50 up to 80 and approaches (1) and (3) by a factor between 80 to 700. On the worst case, when all the processes are interested in receiving the events, they will at most receive them 4 times during 180[s]. This corresponds to 1 duplicate per minute, which is very few.

Figure 20 depicts the number of parasite events received per process.[16] Our algorithm does not induce a lot of parasite events unlike the other two depicted alternatives. Not surprisingly, the more the subscribers, the lesser the parasite events (because more and more subscribers are interested in receiving the events). The greatest number of parasite events received per process is reached when 60% of the processes are interested in receiving such events. In this case, we outperform the other approaches by a factor of 20 to 50 depending on the number of events.

[14] Approach (3) is not shown in this figure because of the high bandwidth it consumes per process (more than 1[MB]).

[15] Again, in Figure 19, we do not show approach (1) and (3) in order to clearly depict the distinction between our algorithm and the best alternative approach (2).

[16] Again, Figure 20 does not contain approach (1), because our algorithm outperforms it by a factor of up to 800 times.

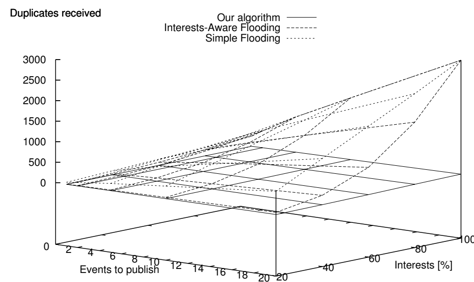

Fig. 19. Number of duplicates received as a function of the number of events to publish and the number of subscribers

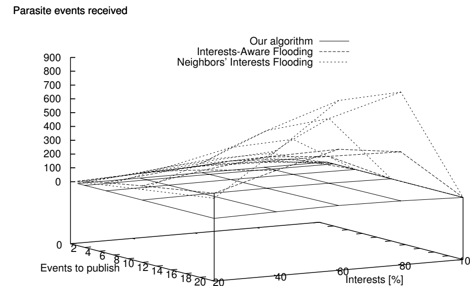

Fig. 20. Number of parasite events received as a function of the number of events to publish and the number of subscribers

6 Concluding Remarks

Many algorithms [10–18] have tackled the issue of disseminating events in a MANET. In [10], the *broadcast storm* problem is introduced. This problem is raised when flooding is used for broadcasting an event in a wireless network. Different schemes are compared: (1) a probabilistic scheme, (2) a counter-based scheme, (3) a distance-based scheme, (4) a location-based scheme and (5) a cluster-based scheme. The last two schemes (i.e., (4) and (5)) rely on a GPS device and cluster heads respectively: assumptions that we do not make in our algorithm. It has been shown in [10] that the first scheme is outperformed by the others. The second and third schemes have been revisited in [19] and feature very interesting characteristics. In our algorithm, we did not explore any distance-based techniques as this would imply more calculation for the mobile devices and require more computing power. In addition, the distance-based scheme together with the counter-based one have been proved to be outperformed by the neighborhood scheme [20]. Our algorithm is close to the latter with certain specificities that we discuss below.

The neighborhood scheme has often been studied in the literature [13–19]. The corresponding algorithms follow roughly one of two different patterns: (1) one-hop neighbor information and (2) multi-hops neighbors information. The first pattern is called self-pruning and the decision of rebroadcasting an event depends on the one-hop knowledge of the neighbors of the processes [18, 13, 19]. This approach achieves fairly good performance without involving too much processing time, which is not the case with the second approach [13, 15–17], where the processes rebroadcast either according to their two-hops neighborhood knowledge [15–17] or according to the decisions of other processes [13]. As the decision of rebroadcasting is often based on a greedy algorithm [21], this consumes a large amount of processing time and is not suited to highly mobile networks. To limit the number of duplicates messages, the neighborhood schemes can be used with a back-off mechanism (like in [14]). In the model we consider, the processes are mobile and only have information about their one-hop neighbors. In this sense, our algorithm belongs to the one-hop category. In our approach however, a process p_i

disseminates an event according to: (1) the validity period of the events of p_i, (2) the subscriptions of the neighbors of p_i and (3) the events those neighbors have received.

The algorithms presented in [11, 12] make specific assumptions on the stabilization of the network, use cluster heads, and switch to flooding when network partitions are frequent. We make no assumption on the topology or stabilization of the network and do not rely on any cluster heads or routing algorithms.

Topic-based pub/sub algorithms for MANETs were also presented in [22–24]. The algorithm relies on brokers which are responsible for buffering the events the subscribers are interested in. When the subscribers connect again to one of the brokers, they ask for the events they have not yet received and the brokers are responsible for providing them with these. Speeding up the bootstrapping latency has been tackled in [25, 26], where client proxies are responsible for collecting events and dispatching them to the real clients when those connect back to the brokers. All these schemes are based on brokers. Our algorithm is completely decentralized.

The approaches described in [27–33] do not rely on brokers. In [27] a direct acyclic graph is maintained between the subscribers and the publishers. To maintain this graph, the network is supposed to remain unpartitioned for some period of time: we do not make this assumption. Moreover, unlike in our algorithm, there can be a huge latency in [27] before a publisher is allowed to publish an event.

A generic way to store data at the most interested mobile processes is described in [28]. The dissemination scheme is not detailed and it is not clear how flooding is avoided when different subscribers have subscribed to the same topic. A specific kind of validity is considered in the sense that each data is associated with a counter which is kept up to date only when the data is used, but the limited memory of the processes is not addressed. In our approach, each event is associated with a timeout that never changes during the entire lifetime of the publication, and after which the event is garbage collected. Like [28], the algorithm presented in [29] implements a distributed hashtable in a MANET. The algorithm of [29] uses dynamic source routing [5] (DSR) to create the routes between publishers and subscribers and consequently floods the network with request and reply messages, which is not the case of our algorithm. Unlike our algorithm, the algorithm of [29] does not consider any validity period for the events and mobile processes must route events they are not interested in. In [30], events are split into several pieces and dispatched on the network. When a process wants to recover the full event, it moves in the network, gathers the different pieces and re-conciliates them. Though this algorithm does not make use of brokers, several processes receive pieces of information they are not interested in, and no notion of validity period is considered.

A pub/sub implementation based on a weakly connected multicast tree is given in [32]. The root of the multicast tree is responsible for publishing the events. This scheme has two drawbacks: the maintenance is time consuming in a high mobile environment and the processes located at the root of the multicast tree have more work to perform than the ones at the leaves. Our algorithm does not need to create or maintain a multicast tree, and processes that have not subscribed to a topic do not need to care about events of that topic.

In the content-based pub/sub algorithm of [33], event dispatchers are responsible for forwarding the events to the interested subscribers and need to store subscriber infor-

mation located multiple hops away. Our algorithm relies only on one-hop information and events are only forwarded by mobile processes that are interested in those.

In the proximity-based algorithm of [31], the subscribers only receive events associated to a certain geographical region. Filtering techniques are used to minimize the burden at publishers and subscribers. In comparison, our algorithm is not limited to a specific location, it supports the dynamic inclusion of topics and exploits the mobility of the processes to disseminate events.

References

1. Cugola, G., Jacobsen, H.A.: Using publish/subscribe middleware for mobile systems. In: Proceedings of the ACM SIGMOBILE Mobile Computing and Communications Review. Volume 6. (2002) 25–33
2. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys **35** (2003) 114–131
3. : (Bluetooth web site: http://www.bluetooth.com/)
4. : (IEEE organisation, 802.11 web site: http://grouper.ieee.org/groups/802/11/)
5. Johnson, D., Maltz, D.: Dynamic source routing in ad hoc wireless networks. In Imielinski, Korth, eds.: Mobile Computing. Volume 353. Kluwer Academic Publishers (1996) 153–181
6. Davies, V.: Evaluating mobility models within an ad hoc network. Master's thesis, Colorado School of Mines (2000)
7. Flury, R., Baehni, S.: EPFL Free Car Parks Application, http://lpdwww.epfl.ch/sbaehni/work/carPark/carPark.html. (2004)
8. Zeng, X., Bagrodia, R., Gerla, M.: Glomosim: a library for parallel simulation of large-scale wireless networks. In: Proceedings of the 12th Workshop on Parallel and Distributed Simulations. (1998)
9. T. Camp, J. Boleng, V.D.: A survey of mobility models for ad hoc network research. In: Proceedings of Wireless Communication and Mobile Computing: Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications. Volume 2. (2002) 483–502
10. Ni, S.Y., Tseng, Y.C., Chen, Y.S., Sheu, J.P.: The broadcast storm problem in a mobile ad hoc network. In: Proceedings of the 5th ACM International Conference on Mobile Computing and Networking. (1999) 151–162
11. Pagani, E., Rossi, G.P.: Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. Journal of Mobile Networks and Applications **4** (1999) 175–192
12. Gupta, S.K.S., Srimani, P.K.: An adaptive protocol for reliable multicast in mobile multi-hop radio networks. In: Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications. (1999) 111–122
13. H. Lim, C.K.: Multicast tree construction and flooding in wireless ad hoc networks. In: Proceedings of the ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems. (2000) 61–68
14. Peng, W., Lu, X.C.: On the reduction of broadcast redundancy in mobile ad hoc networks. In: Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking and Computing. (2000) 129–130
15. Peng, W., Lu, X.C.: AHBP: An efficient broadcast protocol for mobile ad hoc networks. Journal of Science and Technology (2002)
16. Sucec, J., Marsic, I.: An efficient distributed network-wide broadcast algorithm for mobile ad-hoc networks. Technical Report 248, Rutgers University (2000)

17. Qayyum, A., Viennot, L., Laouiti, A.: Multipoint relaying for flooding broadcast messages in mobile wireless networks. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences. (2002) 298–308
18. Cartigny, J., Simplot, D., Carle, J.: Stochastic flooding broadcast protocols in mobile wireless networks. Technical report, LIFL Univ. Lille 1 (2002)
19. Tseng, Y.C., Ni, S.Y., Shih, E.Y.: Adaptive approaches to relieving broadcast storms in a wireless multihop mobile ad hoc network. IEEE Transactions on Computers **52** (2003) 545–557
20. Williams, B., Camp, T.: Comparison of broadcasting techniques for mobile ad hoc networks. In: Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing. (2002) 194–205
21. Lovasz, L.: On the ratio of optimal integral and fractional covers. In: Discrete Mathematics. Volume 13. (1975) 383–390
22. Huang, Y., Garcia-Molina, H.: Publish/subscribe in a mobile environment. In: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access. (2001) 27–34
23. Cugola, G., Nitto, E.D., Fuggetta, A.: The jedi event-based infrastructure and its application to the devlopment of the opss wfms. IEEE Transactions on Software Engineering **27** (2001) 827–850
24. Caporuscio, M., Inverardi, P., Pelliccione, P.: Formal analysis of clients mobility in the siena publish/subscribe middleware. Technical report, Department of Computer Science, University of L'Aquila (2002)
25. Cicila, M., Fiege, L., Haul, C., Zeidler, A., Buchmann, A.P.: Looking into the past: enhancing mobile publish/subscribe middleware. In: Proceedings of the 2nd international workshop on Distributed event-based systems. (2003) 1–8
26. Caporuscio, M., Carzaniga, A., Wolf, A.L.: Design and evaluation of a support service for mobile, wireless publish/subscribe applications. IEEE Transactions on Software Engineering **29** (2003) 1059–1071
27. Anceaume, E., Datta, A.K., Gradinariu, M., Simon, G.: Publish/subscribe scheme for mobile networks. In: Proceedings of the second ACM international workshop on Principles of mobile computing. (2002) 74–81
28. Datta, A., Quarteroni, S., Aberer, K.: Autonomous gossiping: A self-organizing epidemic algorithm for selective information dissemination in wireless mobile ad-hoc networks. In: Proceedings of the International Conference on Semantics of a Networked World. (2004)
29. Pucha, H., Das, S.M., Hu, Y.C.: Ekta: An efficient dht substrate for distributed applications in mobile ad hoc networks. In: Proceedings of the 6th Workshop on Mobile Computing Systems and Applications. (2004)
30. Li, Z., Li, B., Xu, D., Zhou, X.: iFlow: middleware-assisted rendezvous-based information access for mobile ad-hoc application. In: Proceedings of the 2st ACM International Conference on Mobile Systems, Applications and Services. (2003) 71–84
31. Meier, R., Cahill, V.: Steam: Event-based middleware for wireless ad hoc networks. In: Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops. (2002) 639–644
32. Huang, Y., Garcia-Molina, H.: Publish/subscribe tree construction in wireless ad-hoc networks. In: Proceedings of the 4th International Conference on Mobile Data Management. (2003) 122–140
33. Costa, P., Picco, G.P.: Semi-probabilistic Content-Based Publish-Subscribe. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems. (2005)