# Mobility Friendly Publish/Subscribe*

Sébastien Baehni Chirdeep Singh Chhabra Rachid Guerraoui

Distributed Programming Laboratory, EPFL
1015 Lausanne, Switzerland
{sebastien.baehni, chirdeep.chhabra, rachid.guerraoui}@epfl.ch

## Abstract

*This paper describes an event dissemination algorithm that implements a topic-based publish/subscribe abstraction in mobile ad-hoc networks (MANETs). Our algorithm relies on (1) the mobility of the processes and (2) the validity period of the events to ensure the reliability of the dissemination (under reasonable conditions) with a thrifty usage of the memory. The algorithm is inherently portable and does not assume any specific routing protocol. Old events are collected to save the memory and the energy consumption is, in some sense, related to the size of the event scope a subscriber is interested in. We give simulation results in different mobility models and highlight the advantages/drawbacks of our approach as well as we expose some interesting relations between validity periods and reliability.*

## 1. Introduction

The publish/subscribe (pub/sub) communication abstraction is a very appealing candidate for disseminating events in a wireless mobile ad-hoc network (MANET). In such a network, processes are mobile, they may not know each other and might not always be up. With a pub/sub abstraction, remote processes can communicate in a decoupled manner 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 ([11]).

Whereas the writing of MANETs applications is appealing with a pub/sub abstraction, the implementation of such abstraction is not an easy task. In particular, ensuring a reasonable level of reliability of the dissemination is problematic without flooding the entire network. Typically, processes can directly broadcast information in their geographical neighbourhood but need multiple indirections to reach far away processes. In addition, mobile processes do sometimes have a limited memory and the dissemination algorithm cannot use a large portion of it for buffering events. Similarly, the battery power of a process is dynamically limited and a process might not want use it for receiving and forwarding events it is not interested in.

This paper presents an event dissemination algorithm to implement a topic-based pub/sub abstraction in MANETs. Events are arranged according to a topic-hierarchy and subscribing to a specific topic $T_i$, implies being interested in events related to topic $T_i$ and all its subtopics.

Our algorithm ensures that, under reasonable conditions, subscribers receive the events they are interested in with high probability. Events are assumed to have a validity period that depicts the time interval after which they are of no use. Our algorithm exploits this notion together with the mobility of the processes to reliably propagate events to the subscriber processes. Basically, a process keeps propagating events, as long as their validity period have not expired, and as long as there are new neighbours which are interested in the events and did not receive them yet.

As we will discuss in Section 6, several attempts have been made to implement pub/sub abstractions in MANETs. To our knowledge, our algorithm is the first to exploit the mobility of the processes and the validity periods of the events to ensure a level of reliability of the dissemination. Our simulations highlight lower bounds for validity periods to achieve a certain level of reliability, according to the different mobility patterns and processes

* The work presented in this paper was supported 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.

speeds. Our algorithm is inherently portable and does not assume any specific multicast routing protocol (we only rely on a standard wireless layer, e.g., Bluetooth [3], 802.11 [1]). Old events are collected to save the memory and we prevent processes from spending energy in forwarding events they are not interested in (we call these events *noise* events, with respect to the process in question).

The rest of the paper is structured as follows: Section 2 describes our model. Section 3 gives an overview of our algorithm. Section 4 details our pub/sub algorithm. Section 5 gives some simulation performance. Section 6 discusses related work and Section 7 draws some concluding remarks and future directions.

## 2. Model

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

### 2.1. Overview

We assume a set of mobile processes that communicate directly (i.e., one-hop) with their immediate neighborhood.

The architecture depicted in Figure 1(a) is that of a typical pub/sub system with an underlying routing protocol for event dissemination in MANETs, whereas Figure 1(b) describes the architecture we consider: the pub/sub layer lies between the application layer and the MAC (Medium Access Control) layer.

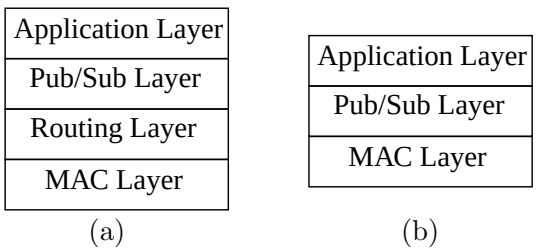



**Figure 1. Pub/Sub architectures.**

### 2.2. Communication Medium

The communication between one process and its neighbors in range is done using the *send* primitive of the underlying MAC layer of the wireless infrastructure. We call the range of a process the geographical zone within which the process can send messages to other processes. It is not possible for a process to send a message to only one of its neighboring processes (i.e., no unicast possible). The communication is one-hop[1] and hence it is not possible for a process to directly *send* a message to processes multiple hops away (no underlying unicast/multicast routing algorithm).

### 2.3. Network Topology

We assume that the network is completely ad-hoc and no fixed infrastructure is present. We moreover do not make any assumption on the connection graph of the processes (i.e., the graph does not need to be fully connected at a given point in time). The processes are assumed to be mobile. We will study two different mobility models: (1) *Random Waypoint* [16] and (2) *City Section* [10], which we explain below.

*Random Waypoint.* In this mobility model, a process moves from its current location to a new location by randomly choosing a direction and speed in which to travel. The new speed and direction are chosen from pre-defined ranges, [*speedmin, speedmax*] and $[0, 2\pi]$ respectively. The Random Waypoint mobility model includes pause times between changes in direction and/or speed [16]. A mobile process stays in one location for a certain period of time (pause time). Once this time expires, the process choses a random destination and moves with a speed as mentioned before.

*City Section.* In the City Section mobility model [10], the simulation area is a street network that typically represents a section of a city where a MANET exists. 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 also common to consider some characteristics like pause times, acceleration and deceleration in certain intersections. This model can be described using a Random Waypoint model with constraints.

### 2.4. 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

1 In the one-hop communication pattern, processes can communicate only with their immediate neighbourhood.

of the range of other processes as well as crash and recover at any time. When two processes can communicate directly (i.e., are in range), we call them neighbors. The neighbourhood of a process $p_i$ is, by extension, the set of processes $p_i$ can communicates directly with. The number of processes in the network is unbounded.

Each event $e_j^{T_k}$ published by a process $p_i$, has (1) a unique identifier $j$, (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., $T_k$.[2] Topics are arranged in a hierarchy (e.g., *.seattle.conferences.mobisys*) and a subscriber that subscribed to a specific topic (e.g., *.seattle.conferences*) is expected to receive events of this topic and all its subtopics (e.g., *.seattle.conferences.mobisys*). The root topic of the topic tree is denoted by the *dot (.)* sign. Finally, an event which topic a process has not subscribed to, is called a *noise* event for that process.

## 3. Algorithm Overview

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

### 3.1. Neighbourhood detection

The processes periodically send heartbeat messages (see Figure 7). These heartbeats contain the following element: (1) the identifier of the process, (2) a list of its subscriptions and, (3) its current speed[3]. The subscriptions are gathered in a list of topics, (e.g.,"$T_i$, $T_j$, $T_k$, ..., $T_n$")[4]. Each process $p_i$ uses the heartbeat messages it receives to construct a dynamic one-hop neighbourhood table, containing the identifiers of the processes in the neighborhood along with their subscriptions and their current speed. Only the processes whose subscriptions intersect with the ones of $p_i$, are kept in $p_i$'s table, other one-hop neighbours are of *no interest* to process $p_i$. The neighbourhood table is continuously garbage collected and updated (depending on the periodicity of the heartbeats). If the speed information is available, the process can adjust the periodicity of the heartbeats. Otherwise, this periodicity is set to a static value (see Section 4.2).[5]

When processes detect each other (with the help of the heartbeat messages), they exchange a list of event identifiers of the events, which are still valid, they have both subscribed to and they already received (i.e., events identifiers of the same topics (or subtopics) of interests). With this information, each process can check if its neighbour is interested in an event it does not already received. If this is the case, the processes proceed to the next phase of the algorithm, namely the dissemination.

### 3.2. Dissemination

When a process detects that one of its neighbours needs an event, it sends the required event to its neighbourhood together with the list of its neighbours, after a back-off period. The calculation of the back-off is presented in details in Section 4.2.

When receiving the event, the neighbouring processes of the sender might decide to send the event again according to a retransmission policy (see Section 4.3). If the processes that receive the events have subscribed to the topic of this event and have not received it yet, they deliver it to the application and store it, till it is garbage collected. If the processes receive events they either already have or have not subscribed to the events' topics, they simply drop the events (in this sense we minimize the burden induced by noise events and save battery power).

### 3.3. Garbage collection

Throughout the two previous phases of our algorithm, we mainly use two main data structures (see Section 4.1) at every process.[6] The first one is used for storing the neighbours that share the same subscriptions as the process itself (neighbourhood table) and the second one is used for storing the events. The neighbourhood table is constantly updated (based on the periodicity of the heartbeats) and its size is upper bounded[7].

---

2 We assume that the event identifier is smaller than the size of the data carried by the event.

3 This information is only useful for optimization purpose and is not mandatory.

4 Remember that subscribing to a topic $T_i$ induces subscription to all its subtopics.

5 We will see in Section 5, having a heartbeat periodicity based on the speed of the processes minimizes the number of messages sent.

6 As we will see, other data structures are involved in the algorithm, but those cannot induce memory problems.

7 The upper bound corresponds to the maximum number of neighbours a process can handle. This bound depends on the structure of the network and on the amount of memory of the processes.

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

### 3.4. Example

Figure 2 depicts a simple scenario illustrating our algorithm. The hierarchy is 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 have been published in the system: $e_3^{T_1}$, $e_4^{T_2}$ and $e_5^{T_2}$. We assume that process $p_1$ has already received $e_3^{T_1}$ and process $p_2$ has already received $e_4^{T_2}$ and $e_5^{T_2}$.

In part I of Figure 2, processes $p_1$ and $p_2$ become neighbours and hence know their common subscriptions. They then exchange the events identifiers corresponding to the topics they have commonly subscribed to and as a consequence, $p_2$ sends to $p_1$ events $e_4^{T_2}$ and $e_5^{T_2}$ (as $T_1$ is a supertopic of $T_2$).

In part II of Figure 2, all three processes become neighbours and exchange their event identifiers and 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. As $p_1$ has more events to send than $p_2$, $p_1$ has a smaller back-off period than $p_2$ (see Section 4.3). $p_1$ then sends events $e_3^{T_1}$, $e_4^{T_2}$ and $e_5^{T_2}$.

In part III of Figure 2, $p_1$ moves on, but $p_2$ and $p_3$ still remain in the range of each other. As $p_2$ was in the range of $p_1$ when it sent the events last, $p_2$ heard the events that $p_1$ sent for $p_3$. Now, $p_2$ and $p_3$ know that they do not have to exchange any more events.

## 4. Algorithm Description

In this section we describe the algorithm in more details focusing first on the data structures involved in the algorithm. Then we describe the neighbourhood detection scheme, the dissemination technique and finally the garbage collection.

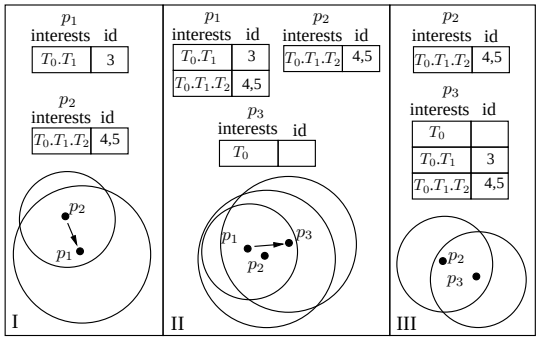



**Figure 2. A simple scenario**

**For each process** $p_i$

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

**Figure 3. Data structures.**

### 4.1. Data Structures

The data structures used in the algorithm are gathered in Figure 3. These consist of a list of *subscriptions* for every process $p_i$ (i.e, $p_i$.subscriptions), a *neighbourhood table* (i.e., *neighbourhoodTable*) and an *event table* (i.e., *eventTable*). These two tables are presented in more details below. There is also the list containing the *events to send* (i.e., *eventsToSend*). Furthermore, different values used in our algorithm are depicted in Figure 3, namely the heartbeat delay (i.e., *HBDelay*), the neighbourhood garbage collection delay (i.e., *NGCDelay*) and the back-off delay (i.e., *BODelay*).

*Subscriptions of a process.* The different subscriptions of every process $p_i$ are stored in a list denoted $p_i$.subscriptions. We assume, without loss of generality, that the size of this list is upper bounded. Indeed, the number of subscriptions of a process is usually limited, and especially in the topic-based scheme (because in this scheme a process only has to subscribe to a topic to receive all the events regarding this topic and all its subtopics).

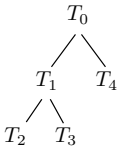
*Neighbourhood Table.* Figure 4 shows the neighbourhood table of a process. The first column of this table stores the identifiers of the neighbours of a process. The second column stores the topics those processes have subscribed to. The third column stores the identifiers of the events the neighbours have received, the forth column contains the speed of the neighbors (this column is not mandatory) and the last column contains the time (i.e., the storetime) when the entry has been stored/updated into the table. This last entry is used to garbage collect the information of the table. We discuss in more details the use of the neighbourhood 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 |

**Figure 4. The neighbourhood table.**

*Event Table.* Each process stores an event table as shown in Figure 5. 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 (i.e., *id*), a validity period (i.e., *validity*), a counter (i.e., *counter*), a topic (i.e., *topic*) and its internal data information (i.e., *data*, this information is not shown in Figure 5). The validity period expresses the time interval after which the event can be removed. The counter represents the number of times an event has been forwarded; it is used, together with the validity period, in the garbage collection sub-protocol, see Section 4.4.

Topic Hierarchy




| 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 |

**Figure 5. The event table.**

**For each process** $p_i$

1: {*The subscription algorithm*}
2: **upon** SUBSCRIBE($T_k$) **do**
3: $p_i$.subscriptions = $p_i$.subscriptions $\cup$ $T_k$
4: **if** (HEARTBEAT not started) **then**
5: start HEARTBEAT
6: **end if**
7: **if** (NEIGHBOURHOODGC not started) **then**
8: start NEIGHBOURHOODGC
9: **end if**
10: **end upon**

11: {*The unsubscription algorithm*}
12: **upon** UNSUBSCRIBE($T_k$) **do**
13: $p_i$.subscriptions = $p_i$.subscriptions $\setminus$ $T_k$
14: **if** ($p_i$.subscriptions == $\emptyset$) **then**
15: stop HEARTBEAT
16: stop NEIGHBOURHOODGC
17: **end if**
18: **end upon**

**Figure 6. Subscription, Unsubscription.**

*The events to send.* This structure contains the events a process will send to its neighbours. This structure can be, at most, as big as the event table (if a process has to send all its events to its neighbours). However, this structure is always reset each time the events are sent (i.e., after each back-off). The size of this structure is upper bounded and it is reset periodically.

## 4.2. Neighbourhood Detection

Before being able to detect neighbours, the processes have to subscribe to topics they are interested in. The subscription/unsubscription sub-protocol is given in Figure 6. 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 *neighbourhoodGC*[8] 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 *neighbourhoodGC* tasks are stopped.

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. Note that the information about the speed of the processes is not mandatory and is only used, as an optimization, to reduce the number of heartbeats messages (see Section 4.2). After receiving the heartbeat messages, each process is able to

8 The *neighbourhoodGC* task is used for garbage collecting the neighbours from the neighbourhood table and is presented in more details in Section 4.4.

**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 neighbourEvent(j,subscriptions)
 9:       if (j ∉ neighbourhoodTable) then
10:          neighbourhoodTable ∪
             (j,subscriptions,[speed],currentTime)
11:       else
12:          UPDATENEIGHBOURINFO(j,subscriptions,[speed],
             currentTime)
13:       end if
14:    end if
15:    COMPUTEHBDELAY(neighbourhoodTable)
16:    COMPUTENGCDELAY(neighbourhoodTable)
17:    COMPUTEBODELAY(neighbourhoodTable)
18: end upon

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

25: {Reception of a list of events identifiers}
26: upon RECEIVE(j, eventsIDs) do
27:    if j ∈ neighbourhoodTable then
28:       for all eventID ∈ eventsIDs do
29:          UPDATENEIGHBOUREVENTINFO(j, eventID, cur-
             rentTime)
30:       end for
31:       RETRIEVEEVENTSTOSEND()
32:    end if
33: end upon
```

**Figure 7. Neighbourhood detection.**

build a view of its neighbourhood, together with a list of their subscriptions. If two neighbouring processes do not share any common topics, these topics are not stored in their respective neighbourhood table. The neighbourhood information of a process is stored in the table presented in Figure 4 and updated accordingly (using the UPDATENEIGHBOURINFO() method[9]).

If the subscriptions of a process match the ones of its neighbour, they then exchange the events identifier they respectively have subscribed to (the events identifiers are retrieved via the GETEVENTSIDS() method). Once those events identifiers are received, the process updates its neighbourhood table with those and checks if it has to send events to its neighbour (via the RETRIEVEEVENTSTOSEND() method, see Section 4.3). The events identifiers are exchanged instead of the plain events for minimizing the size of the duplicates messages. It may happen that a process and its neighbour, has the same set of events; in this case, there is no use for them to exchange the events. The only possibility for preventing this situation is by exchanging events identifiers before sending the real events.

9 This method is not described in the algorithm for space limitations. However it simply consists of updating the information (i.e., subscriptions, speed and store time) corresponding to the right neighbour.

**For each process $p_i$**

```
 1: {Computation of the hearbeat delay}
 2: function COMPUTEHBDELAY(neighbourhoodTable)
 3:    averageSpeed =
       AVERAGESPEED(neighbourhoodTable)
 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 neighbourhood garbage collection
    delay}
11: function COMPUTENGCDELAY(neighbourhoodTable)
12:    NGCDelay = HBDelay*HB2NGC
13: end

14: {Computation of the back-off delay}
15: function COMPUTEBODELAY(neighbourhoodTable)
16:    BODelay = HBDelay / (HB2BO*sizeof(eventsToSend))
17: end
```

**Figure 8. Computing delays.**

The computation of the time intervals for (1) the heartbeat messages, (2) the neighbourhood garbage collection and (3) the back-off period are determined at the reception of the heartbeat messages, via respectively the (1) COMPUTEHBDELAY(), (2) COMPUTENGCDELAY() and (3) COMPUTEBODELAY() methods. Figure 8 gives a possible implementation of these methods.

## 4.3. Dissemination

Our dissemination algorithm is described in Figure 10. Basically the process uses the PUBLISH() method to send the event to the neighbouring processes if at least one of those has subscribed to the topic of the event. Moreover, in calling this method, the process updates the neighbour information in its neighbourhood table (via the UPDATENEIGHBOUREVENTINFO() method[10]).

10 For space limitation this method is not shown in the algo-

**For each process $p_i$**

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

**Figure 9. Retrieval of the events.**

As soon as a process receives an event, it updates its neighbourhood table (using the UPDATENEIGHBOUREVENTINFO() method) with the list of neighbour identifiers it 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 neighbours (i.e., RETRIEVEEVENTSTOSEND() method, Figure 9).

If a process $p_i$ finds out that some of its neighbours have subscribed to the topic of the events $p_i$ owns, $p_i$ starts a back-off period (the back-off delay is determined by the function COMPUTEBODELAY()[11]). Taking into account the events that have been received by the processes reduces the number of useless retransmissions and hence saves power and bandwidth.

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

rithm; it basically consists in updating the list of events a neighbour is supposed to have received with its event identifier.

11 A possible implementation of this method is shown in Figure 8. In this implementation, the back-off delay depends on the heartbeat delay and on 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:    RETRIEVEEVENTSTOSEND()
4:    if eventsToSend ≠ ∅ then
5:       SEND(i, eventsToSend, neighboursIDs)
6:       eventsIDs = GETEVENTSIDS(eventsToSend)
7:       for all neighbourID ∈ neighbourhoodTable do
8:          for all id ∈ eventsIDs do
9:             UPDATENEIGHBOUREVENTINFO(neighbourID,
               id)
10:         end for
11:      end for
12:      INCREMENT(eventsToSend, eventsTable)
13:   end if
14: end upon

15: {Publication of a new event e_k^{T_j}}
16: function PUBLISH(i, e_k^{T_j}, neighboursIDs)
17:    for all neighbour ∈ neighbourhoodTable do
18:       if neighbour.subscriptions ∈ p_i.subscriptions then
19:          interested = true
20:          break
21:       end if
22:    end for
23:    if interested then
24:       SEND(i, e_k^{T_j}, neighboursIDs)
25:       for all neighbourID ∈ neighbourhoodTable do
26:          UPDATENEIGHBOUREVENTINFO(neighbourID, k)
27:       end for
28:    end if
29: end

30: {Reception of a list of events}
31: upon RECEIVE(j, events, neighboursIDs) do
32:    for all e_k^{T_j} ∈ events do
33:       for all neighbourID ∈ neighboursIDs do
34:          UPDATENEIGHBOUREVENTINFO(neighbourID, k)
35:       end for
36:       if T_j ∈ p_i.subscriptions then
37:          interested = true
38:          STOP backOff timer
39:          if eventsTable is full then
40:             garbageCollect(eventsTable)
41:          end if
42:          eventsTable ∪ e_k^{T_j}
43:          DELIVER(e_k^{T_j})
44:       end if
45:    end for
46:    if interested then
47:       RETRIEVEEVENTSTOSEND()
48:    end if
49: end upon
```

**Figure 10. Dissemination.**

## 4.4. Garbage Collection

We collect old information both in the neighbourhood table and in the event table.

**For each process** $p_i$

```
1: {Garbage collection of the neighbourhood table}
2: task neighbourhoodGC
3:    for all neighbour ∈ neighbourhoodTable do
4:       if currentTime - NGCDelay > neighbour.storeTime
         then
5:          REMOVE(neighbour,neighbourhoodTable)
6:       end if
7:    end for
8: end

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}) / (fwd(e_k^{T_j})+val(e_k^{T_j})) ≤ val(gc) / (fwd(gc)+val(gc))
         then
14:          gc = e_k^{T_j}
15:       end if
16:    end for
17:    REMOVE(gc,eventsTable)
18: end
```

**Figure 11. Garbage collection.**

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

*Neighbourhood table.* Each time the neighbourhood garbage collection delay expires, all the process identities which storetimes have expired, are collected from the neighbourhood table (see Figure 11). As this task is executed periodically and as we can assume that the total number of simultaneous neighbours is limited, the size of the table is upper bounded.

*Event table.* As our algorithm exploits the validity period of the events, those are simply collected when their validity period expires. However, besides the validity parameter, the number of times an event was propagated is also considered in order to best propagate all the events. As a matter of example, events with high validity periods that have been propagated several times have to be collected before events with short validity periods that have never been forwarded, otherwise these events will never be disseminated in the system. Equation 1 captures the way we collect the events, based on their: (1) validity period (i.e., val($e_k^{T_j}$)) and (2) the number of times an event has been forwarded (i.e., fwd($e_k^{T_j}$)). This equation ensures that if an event with a high validity period has been sent a certain number of times, it is collected before an event with a short validity period that has never or rarely been forwarded. The garbage collection function for an event $e_k^{T_j}$ is given as ($\forall val(e_k^{T_j}), fwd(e_k^{T_j}) \in \mathbb{N}^*$):

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

In Figure 12, we present the shape of the garbage collection function (i.e., gc($e_k^{T_j}$)), for different validity values and different forwarding values[12]. When the event table is full, we remove the event with the lowest garbage collection value (Figure 12).

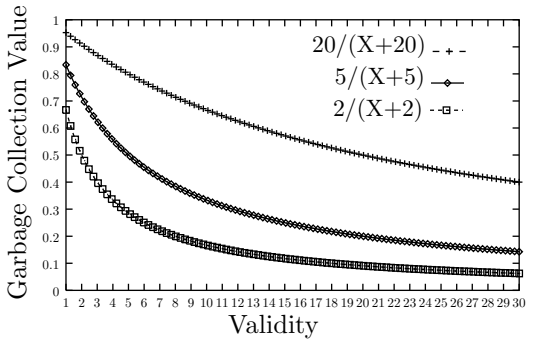



**Figure 12. The garbage collection function**

As an example, we can see in Figure 12, that an event with a validity period of 2 that have been forwarded less than 2 times, will be collected after an event with a validity period of 5 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 last structure is efficiently garbage collected, hence the size of the events to send structure cannot grow indefinitely.

## 5. Performance

In this Section, we present the performance results obtained from simulating our algorithm according to various mobility patterns. Basically, we show that, for different mobility patterns, it is possible to achieve the same reliability while varying the speed of the processes and the validity of

12 This information is available using the events table (i.e., counter column)

the events. We first describe the simulator we use, present the overall settings and finally we give the actual performance measurements.

### 5.1. Environment

We simulated our pub/sub algorithm using *Qualnet* (3.7) [28], directly on the 802.11b MAC layer, in two different mobility models, : (1) *Random Waypoint* [16] and (2) *City Section* [10].

### 5.2. Experimental Settings

We first give the settings underlying our algorithm implementation and then the settings underlying Qualnet for the different mobility pattern considered.

*Our algorithm.* Regarding the common configuration parameters of our algorithm, we have set the size of the events to 400 bytes, *x* to 40, *HB2BO* to 2 and *HB2NGC* to 2.5. The upper bound value for the heartbeat period varies[13], as well as the mobility of the processes and the validity of the events (see the following performance measurements configuration). The choice of these values (i.e., *x*, *HB2BO* and *HB2NGC*) can be subject to discussion and, as usual, is a matter of trade off between the overall number of messages sent (heartbeats, events identifiers, and actual events) and the reliability of receiving those (as we will see in Section 5.3). Moreover, for the special case of the Random Waypoint model, we have run the experiments after the first 600 seconds of the simulation time (due to the high variability in the neighbourhood percentage during these first seconds [25]).

*Random Waypoint in Qualnet.* The Qualnet simulator let the user choose between the following properties: (1) the minimum speed of the processes, (2) the maximum speed of the processes and (3) the pause time between each destination transition. In our experiments, the pause time has always been set to 1[s]. The maximum and minimum speed vary during the entire set of experiments, see Section 5.3. Moreover, in this model, we have done our experiments on a virtual area of 25[$km^2$], populated randomly with 150 processes.

Regarding the overall settings of the simulator, we have used a "standard" 802.11b ad-hoc network. We have used the default values provided by the Qualnet simulator. Namely, the transmission power rate is of 15[db] for all the 1,2,6 and 11[Mbps] and the reception sensitivity is -93[db], -89[db], -87[db] and -83[db] for 1,2,6 and 11[Mbps] respectively.[14] The channel frequency is 2.4[Ghz] and it uses a statistical propagation model, with a limit of -111[dbm] and a two ray path loss model. The MAC protocol used is the 802.11 one. Each process has an omni-directional antenna with an efficiency of 0.8.

*City Section in Qualnet.* Qualnet let also the user define its own mobility patterns for each process in the system and hence allows to simulate a city with its traffic. For this model, we have taken the map of our campus at EPFL and created a specific mobility pattern for each of 15 processes. The EPFL campus covers 1200x900 square meters. The processes do not walk/drive randomly in each of the roads. We have tried to follow the real traffic conditions, in the sense that some roads are more often used than others.

Regarding the overall settings of the simulator, they are the same as for the Random Waypoint, except for the reception sensitivity which is of -65[db] for all rates (1,2,6 and 11[Mbps])[15]. We have modified these values to try to simulate the real radio range in a city.

### 5.3. Results

We first focus on the Random Waypoint model and then on the City Section model.

*Random Waypoint Model.* We have conducted the simulation for different speed values: 0[mps], 1[mps], 5[mps], 10[mps], 20[mps], 30[mps] and 40[mps]. All the simulation were ran 30 times with different seed values and the results presented in each figure are averages of the 30 obtained values. There is only one event published by one original publisher.

In the first experiment, each process has subscribed to the topic of the event. This event has a validity of 60[s]. We have varied the speed of each process as well as the heartbeat upper bound limit (15[s], 10[s], 5[s] and 1[s]). Not surprisingly, more mobility implies more reliability. Regarding the upper bound limit, we can see that if we try to have a small upper bound, the better the reliability, even in networks with low mobility. It is interesting to notice that sending heartbeats often in a low mobile network, in our algorithm, increases the reliability. This can be explained by the fact that a message loss is more critical in a network where the

13 The heartbeat upper bound determines the upper bound of the heartbeat period.

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

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

heartbeat periods are long. Indeed, the propagation of the event to other processes is much longer and, as the validity is low (60[s]), the probability that all processes receive the event is also low. However, sending more often those heartbeats (and consequently the events identifiers messages), has a cost: the number of total messages sent with a heartbeat upper bound of 1[s] is 2.5 more than with a heartbeat upper bound of 15[s].

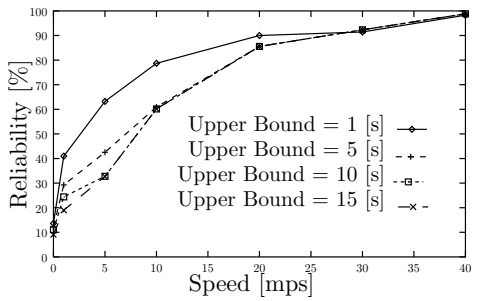



**Figure 13. Probability of event reception as a function of the heartbeat period.**

In the second experiment, we consider the validity of the events and the speed of the processes (the heartbeat period is set to 1[s]). The plain and dashed graph represent reliability values obtained when only 20%, respectively 80%, of processes have subscribed to the topic of the event. We can see that, when a few processes have subscribed to the topic of the events, it is very difficult to achieve high reliability. Only a combination of a high mobility of the processes and a high validity of the events can lead to a fairly good reliability. For instance, when the processes move at a speed of 30[mps], events with a validity of 150[s] are received by 75% of the subscribers. We can explain this with the fact that the simulating 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, this means that we have only 30 processes for a region of 25[$km^2$]; the network is too sparse.

However, when more processes have subscribed to the topic of the events (80%), we can achieve a fairly high reliability with different validity values and different speed of the processes. For example, processes moving at 10[mps] and publishing events of 180[s] validity, have the same 95% reliability than processes moving at 30[mps] and publishing events which validity is 90[s]. Interestingly, it is hence possible, according to different mobility patterns and speed of the processes, to give the lowest validity value for achieving a specific reliability.

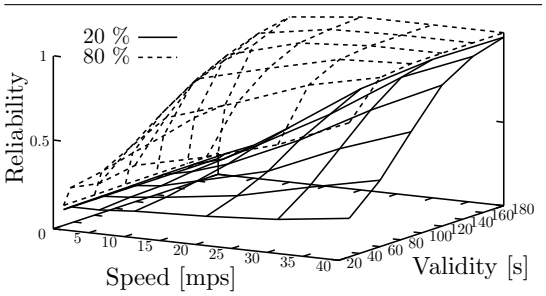



**Figure 14. Probability of event reception as a function of the validity, the speed of the processes and the number of subscribers.**

In Figure 15, 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 subscriber number, the reliability is low also. However, even if only 60% of the processes have subscribed to the topic of an event with a validity of 120[s], then almost all of them will receive it. We can relate these results with the ones of a network in which all processes move at a speed of 20[mps]. Indeed, according to our results the overall reliability of reception depends on the average speed of all the processes in the network rather than on the specific speed of each process.

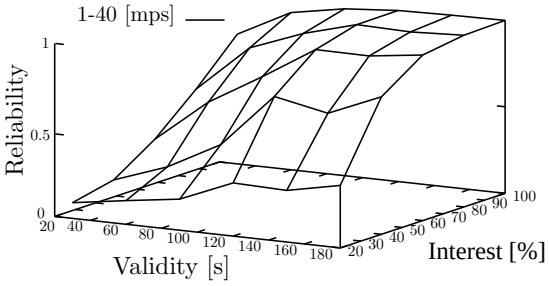



**Figure 15. Probability of event reception as a function of the validity and the number of subscribers, in a heterogeneous mobile environment.**

*City Section Model.* As mentioned previously, the simulation is composed of 15 processes driving in

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

**Figure 16. 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% |

**Figure 17. Probability of event reception as a function of number of subscribers.**

a 1200 x 900 [$m^2$]. All processes walk/drive at a given speed which is the speed limit of the road they are currently walking/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, ...). In all the different experiments, all processes, in turn, become the original publisher. Again all experiments have been conducted 30 times and results we present are an average over these 30 times on all the 15 publishers.

In the first set of experiments, we also tested the importance of the heartbeat period over the overall reliability. Recall that, in such a network, with the upper settings and 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 16 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 of this event is 150[s]. The presented results are an average on the values obtained for all the processes being once the original publisher.

One of the conclusions of our experiments is that there is not a real difference of reliability between the heartbeats sent every 1[s] or 2[s]. However, between 1[s]-2[s] and 5[s], we have a lost of 22% reliability, which can be quite important. An interesting think to notice is the fact that having heartbeats every 4[s] is better than having them every 3[s]. This surprising result can be explained by the fact that, with this heartbeat period of 3[s], the messages sent by the processes are more likely to collide than with the predefined mobility environment. Hence, it may happen that, given a mobility environment, a heartbeat periodicity both combining the interval delay and the reliability can be found and that this periodicity might not be the smaller one. We are still investigating this area.

In the second set of experiments, we have set the heartbeat upper bound period to 1[s] and we have varied the number of subscribers from 20% to 100%. Again, the values presented in Figure 17 are an average on all the processes being in turn the original publisher.

Interestingly, these results are not comparable to the ones obtained in the Random Waypoint model. Even if only 20% of the processes have subscribed to the topic of the event, almost 60% of them will receive the event which was not the case in the previous model. Again, this can be explained by the fact that, in this model, the processes do not move randomly and follow specific paths defined according to specific rules. Indeed, they are more likely to become neighbours than in the Random Waypoint model, especially if several roads have more importance than others (which is the case in our simulations).

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

**Figure 18. Difference of reliability between the processes.**

We also point out the importance of the path of the processes when we compare the reliability achieved according to each of the publishers specifically. In Figure 18, 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.

In the final set of experiments, we have set the heartbeat upper bound period to 1[s] and we have made vary the validity of the events from 20[s] to 150[s]. Figure 19 shows the results.

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

**Figure 19. Probability of event reception as a function of the event validity.**

On the contrary of the subscriber percentage, we can see that the validity 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* the processes become neighbours with *when* they become neighbours. In the Random Waypoint model, the processes exchange information uniformly during the simulation, there is no real hot point where the processes meet. Whereas in the City Section model, the processes meet and exchange their information more likely at social meeting points, hence the huge differences in reliability.

## 6. Related Work

This section relates our work to alternative ways to implement pub/sub in MANETs. We first discuss some general purpose dissemination algorithms which could be used to implement pub/sub in MANETs. Then we focus on algorithms specifically designed with MANETs pub/sub in mind.

### 6.1. Dissemination algorithms

Many algorithms ([19, 20, 12, 13, 21, 22, 24, 23, 6]) tackle the issue of disseminating events in a MANET. In [19], the *broadcast storm* problem is introduced, 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 on cluster heads respectively: assumptions that we do not make in our implementation. It has been shown in [19] that the first scheme is outperformed by other algorithms (like the counter-based scheme). The second and third schemes have been revisited in [26] and present very interesting characteristics. We however do not focus on the distance-based technique as it implies more calculation for the mobile devices and requires more computing power. Moreover, the distance-based scheme together with the counter-based one are outperformed by the neighbouring scheme [27]. Our pub/sub algorithm is close to this last scheme with several modifications we discuss in the following.

The neighbouring scheme has been often studied ([6, 13, 21, 22, 24, 23, 26]). These algorithms follow roughly one of two different patterns: (1) one-hop neighbour information and (2) two-hops neighbours information. The first pattern is called self-pruning and the decision of rebroadcasting an event depends on the one-hop knowledge of the neighbours of the processes ([6, 13, 26]). This approach achieves fairly good performance while not taking too much processing time, which is not the case with the second approach ([13, 22, 24, 23]), where the processes rebroadcast either according to their two-hops neighbourhood knowledge ([22, 24, 23]) or according to the decisions of other processes ([13]). As the decision of rebroadcasting is often based on a greedy algorithm ([17]), this decision takes processing time and is not suited in high mobile networks. To limit the number of duplicates messages, the neighbouring schemes can be mixed with a back-off mechanism (like in [21]). In the model we consider, the processes are mobile and only have information about their one-hop neighbours. Basically this means that our algorithm belongs to the one-hop category. In our approach, a process $p_i$ disseminates an event according to: (1) the validity of the events of $p_i$, (2) the subscriptions of the neighbours of $p_i$ and (3) the events those neighbours have received.

Algorithms like in [20, 12] rather make 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 network (except that the processes are mobile) and do not rely on cluster heads.

### 6.2. Publish/Subscribe in Mobile Ad-Hoc Networks

In [14, 8], the authors present a topic-based pub/sub algorithm for MANETs. The algorithm relies on brokers, like in Siena [5], in which each broker is 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. Speeding up the bootstrapping latency has been tackled in [7, 4], where client proxies are responsible for collecting the events and dispatch them to the real clients when those connect back to the brokers. All these schemes are based on brokers and enhance them to support client mobility whereas our algorithm in completely decentralized.

The approaches described in [2, 9, 18, 15] do not rely on brokers. In [2] 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.[16]

16 Moreover, in [2], there can be a huge latency before a publisher is allowed to publish an event, which is not our case.

A generic way to store data at the most interested mobile processes is described in [9]. The dissemination scheme is not detailed and it is not clear how flooding is avoided when different subscribers have subscribed in the same content. 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 if the data is used, but the authors do not take into account the limited memory of the processes. 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.

A pub/sub implementation based on a weakly connected multicast tree is given in [15]. 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 than the ones at the leaves. Our approach does not need to create and maintain a multicast tree. In addition, processes that have not subscribed in a topic do not need to forward events of that topic.

In [18], a proximity-based algorithm is described. The subscribers only receive events associated to a certain geographical region. Filtering techniques to minimize the burden at publishers and subscribers are used. Our approach is not limited to a specific location, supports the inclusion of topics and exploits the mobility of the processes to disseminate events.

## 7. Concluding remarks

This paper presents an event dissemination algorithm to implement a topic-based publish/subscribe abstraction in MANETs. Our algorithm is the first to exploit the mobility of the processes and the validity periods of the events to ensure a reasonable high level of reliability of the dissemination, while making a thrifty usage of the memory and the battery power. It is inherently portable and it prevents processes from spending energy on forwarding events they are not interested in. Our performance measures highlight a lower bound on the validity period of an event in order to ensure a reliable dissemination according to the chosen mobility pattern.

## References


[1] IEEE organisation, 802.11 web site (http://grouper.ieee.org/groups/802/11/).

[2] E. Anceaume, A.-K. Datta, M. Gradinariu, and G. Simon. Publish/subscribe scheme for mobile networks. In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pages 74–81, 2002.

[3] Bluetooth web site (http://www.bluetooth.com/).

[4] M. Caporuscio, A. Carzaniga, and A.-L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec 2003.

[5] M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of clients mobility in the siena publish/subscribe middleware. Technical report, Department of Computer Science, University of L'Aquila, October 2002.

[6] J. Cartigny, D. Simplot, and J. Carle. Stochastic flooding broadcast protocols in mobile wireless networks. Technical report, LIFL Univ. Lille 1, may 2002.

[7] M. Cicila, L. Fiege, C. Haul, A. Zeidler, and A.-P. Buchmann. Looking into the past: enhancing mobile publish/subscribe middleware. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, 2003.

[8] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the devlopment of the opss wfms. *IEEE Transactions on Software Engineering*, 27(8):827–850, september 2001.

[9] A. Datta, S. Quarteroni, and K. Aberer. 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 (IC-SNW'04)*, 2004.

[10] V. Davies. Evaluating mobility models within an ad hoc network. Master's thesis, Colorado School of Mines, 2000.

[11] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[12] S. K. S. Gupta and P. K. Srimani. 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*, pages 111–122, february 1999.

[13] C. Kim H. Lim. 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 (MSWIM)*, pages 61–68, 2000.

[14] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34, 2001.

[15] Y. Huang and H. Garcia-Molina. Publish/subscribe tree construction in wireless ad-hoc networks. In

*Proceedings of the 4th International Conference on Mobile Data Management*, pages 122–140, 2003.

[16] D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353, pages 153–181. Kluwer Academic Publishers, 1996.

[17] L. Lovasz. On the ration of optimal integral and fractional covers. In *Discrete Mathematics*, volume 13, pages 383–390, 1975.

[18] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, pages 639–644, 2002.

[19] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th ACM International Conference on Mobile Computing and Networking*, pages 151–162, 1999.

[20] E. Pagani and G. P. Rossi. Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. *Journal of Mobile Networks and Applications*, 4:175–192, 1999.

[21] W. Peng and X.-C. Lu. 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*, pages 129–130, 2000.

[22] W. Peng and X.-C. Lu. Ahbp: An efficient broadcast protocol for mobile ad hoc networks. *Journal of Science and Technology*, 2002.

[23] A. Qayyum, L. Viennot, and A. Laouiti. Multipoint relaying for flooding broadcast messages in mobile wireless networks. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, pages 298–308, january 2002.

[24] J. Sucec and I. Marsic. An efficient distributed network-wide broadcast algorithm for mobile ad-hoc networks. Technical Report 248, Rutgers University, september 2000.

[25] V. Davies T. Camp, J. Boleng. A survey of mobility models for ad hoc network research. In *Proceedings of Wireless Communication and Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications, vol 2*, pages 483–502, September 2002.

[26] Y.-C. Tseng, S.-Y. Ni, and E.-Y. Shih. Adaptive approaches to relieving broadcast storms in a wireless multihop mobile ad hoc network. *IEEE Transactions on Computers*, 52(5):545–557, may 2003.

[27] B. Williams and T. Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 194–205, march 2002.

[28] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations (PADS '98)*, May 1998.