Publish-Subscribe on Sensor Networks: A Semi-probabilistic Approach

Paolo Costa, Gian Pietro Picco
Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
E-mail: {costa,picco}@elet.polimi.it

Silvana Rossetto
Departamento de Informatica
PUC-Rio, Rio de Janeiro, Brazil
E-mail: silvana@inf.puc-rio.br

Abstract

In this paper we propose a routing strategy for enabling publish-subscribe communication in a sensor network. The approach is semi-probabilistic, in that it relies partly on the dissemination of subscription information and, in the areas where this is not available, on random rebroadcast of event messages. We illustrate the details of our approach, concisely describe its implementation in TinyOS [14] for the MICA2 platform [1], and evaluate its perfomance through simulation. Results show that our approach provides good delivery and low overhead, and is resilient to connectivity changes in the sensor network, as induced by the temporary standby necessary to preserve the energy of sensor nodes.

1. Introduction

The miniaturization of computing, sensing, and wireless communication devices recently enabled the development of *wireless sensor networks* (WSN), a new form of distributed computing where sensors deployed in the environment communicate wirelessly to gather and report information about physical phenomena. Several successful applications of WSNs are reported in the literature [2, 3, 5].

A fundamental issue in realizing a WSN is how to route the applicative data, i.e., the messages controlling the operation of the various sensors and the data gathered by them. Most of the existing approaches assume the existence of a single data sink—typically, a centralized monitoring station— interested in the sensed data and focus on optimizing multi-hop communication among sensors to route messages efficiently to and from the sink. In general, however, multiple data sinks may exist in the system, potentially interested in monitoring different phenomena whose behavior however can be derived by analyzing the same set of raw sensed data. This is evident in the case where multiple monitoring stations, possibly mobile as in [10], are deployed in the system. However, it is even more poignant in a variation of WSN that is rapidly attracting interest among researchers and practitioners, namely, *wireless sensor and actor networks* (WSAN) [4]. In this case, the devices deployed in the environment are not only able to sense environmental data, but also to react by affecting the environment with their actuators. However, to do so they usually play the role of data sinks, in that they rely on the data sensed and disseminated by the other devices in the network.

Despite the rapid development of this research field, the state of the art shows how programming sensor network applications is still done by and large in an ad hoc fashion. As usual, software evolves slower than hardware, and although the first middleware and platforms for sensor networks are beginning to appear (e.g., [13, 20, 23]), most of the efforts are still devoted to the core OS and network functionality, with little attention to higher-level abstractions that simplify distributed programming without sacrificing performance.

In this context, the publish-subscribe interaction paradigm naturally resonates with sensor networks. Publish-subscribe middleware is organized as a collection of *client* components, which interact by *publishing* messages and by *subscribing* to the classes of messages they are interested in. The core component of the middleware, the *dispatcher*, is responsible for collecting subscriptions and forwarding messages from publishers to subscribers. In sensor network context, for instance, an actuator may be interested in receiving all the messages concerning a temperature greater than 30 degrees, to activate a fan; similarly, a node hosting a temperature sensor may subscribe to all the messages carrying application queries for temperature data. The implicit and asynchronous communication paradigm that characterize publish-subscribe fosters a high degree of decoupling among the components, which is beneficial since the system configuration often changes as the devices enter power saving mode.

Clearly, the difficulty is how to implement efficient routing strategies for a distributed dispatcher implementation. Our research group has been recently very active in tackling this problem in contexts with a dynamic topology, including MANETs (see e.g., [8, 19]). In particular, we recently de-

vised a routing strategy [9] that exploits a semi-probabilistic approach. Message subscriptions are propagated deterministically only in the immediate vicinity (in terms of number of hops) of the subscribing node. When a message is published, it is routed using this deterministic information—if available. If there is no such information to determine the next hop, the decision is taken probabilistically, by forwarding the message along a randomly selected subset of the available links. Being based on probabilistic decision, our approach exhibits very low overhead, but cannot guarantee 100% delivery in all situations. Nevertheless, it is geared towards highly dynamic scenarios where the cost of providing full delivery guarantees are prohibitive—when a solution exists. The simulations in [9] confirm that the approach performs well (i.e., high delivery and low overhead) even in very dynamic scenarios, and better than a purely probabilistic (or deterministic) approach.

In this paper, we start from the same premises of employing a semi-probabilistic approach. Its characteristics of low overhead and resilience to changes in the network topology make it amenable to sensor networks, where in many cases (e.g., continuous monitoring) probabilistic guarantees are enough. Nevertheless, in this paper we tailor our original solution to the peculiar characteristics of our new target scenario. First of all, we adopt a different communication model. In [9] we assumed communication to take place along the links of a graph-shaped overlay network: here, instead, the broadcast facility provided by sensors is our only communication media. Moreover, the overlay network completely masked the mechanics of the underlying network communication: here, instead, by relying directly on wireless broadcast we need to take into account packet collisions, to avoid depleting the sensors' power on useless retransmissions. Finally, sensors often operate in a duty cycle, by alternating processing and communication with stand-by periods, therefore saving battery power. This introduces a particular form of dynamicity in the network, even in absence of mobility. To evaluate our routing strategy we implemented it on Crossbow's MICA2 *motes* running TinyOS [14], and emulated its behavior with TOSSIM [16] in scenarios with up to 400 nodes. The research contribution of this paper is therefore twofold. First, we extend, adapt, and evaluate our semi-probabilistic approach for broadcast communication in the context of sensor networks. Second, our implementation can be effectively regarded as a novel publish-subscribe middleware for sensor networks.

The paper is organized as follows. Section 2 presents the details of our approach, while Section 3 concisely describes its TinyOS implementation. Section 4 reports about the evaluation of our approach using the `tossim` emulator in several scenarios. Section 5 places our work in the context of related efforts. Finally, Section 6 ends the paper with brief concluding remarks.

2. Approach

In this section we provide a complete, albeit informal, description of our approach. In the following we assume wireless broadcast is the only communication media used, and also assume that each (active) sensor takes part in routing, regardless of whether it is currently interested in publishing and/or subscribing. Finally, we observe that a distinction is usually drawn between *subject-based* systems, where subscriptions are specified by selecting a topic among many defined a priori, and *content-based* systems, where instead subscriptions are defined using *filters* over the actual message content. Content-based publish-subscribe systems are much more expressive, but often demand a more complex implementation[1]. In the specific case of our approach, the difference is entirely confined in the format of the subscription message, and therefore both variants of publish-subscribe can be implemented equally easily.

2.1. Disseminating and Managing Subscriptions

The subscriptions issued by the application components disseminate in the network deterministic information that is going to be used for routing events. When the application running on a node issues a subscription, our middleware broadcasts the corresponding filter. This information is rebroadcast by the subscriber neighbors to an extent defined by the *subscription horizon* ϕ. In our original, link-based approach [9], ϕ was measured as the number of hops travelled by a subscription message along the links of the graph overlay. In this paper, instead, ϕ represents the number of times the subscription message is (re)broadcast. A value $\phi = 0$ means that no subscription is ever transmitted by the subscriber node, and therefore the corresponding information is only stored locally in the subscription table. As we discuss next, this implies that events are routed in a purely probabilistic fashion. If $\phi > 0$, the subscriber broadcasts the subscription; the neighbors receiving the message update their subscription table accordingly. If $\phi = 1$, no further action is taken. Otherwise, the subscription is rebroadcast by the neighbors to the extent mandated by ϕ.

In a publish-subscribe system, subscriptions can be issued and removed dynamically by using proper middleware constructs, to reflect the changing interests of applications. Clearly, the information held by the middleware infrastructure, and in particular the content of the subscription tables, must be updated accordingly. In [9], we exploited the standard technique of dealing with (un)subscriptions explicitly, by using control messages propagated whenever a node decides to (un)subscribe. The same technique is used to deal with appearing or vanishing links, by treating the disap-

1 See [12] for a comparison and more detailed discussion.

pearing endpoint as if it were, respectively, subscribing or unsubscribing. Here, we use a different strategy that associates *leases* to subscriptions, and require the subscriber to refresh subscriptions by re-propagating the corresponding message[2]. If no message is received before a lease expires, the corresponding subscription is deleted.

Clearly, there are tradeoffs involved. Without a leased approach the (un)subscription traffic is likely to be significant, due to the need to reconcile routing information whenever a link appears or disappears. The leased approach remarkably reduces the communication overhead, by removing this need. On the other hand, if subscriptions are stable, bandwidth is unnecessarily wasted for refreshing leases. However, in sensor networks the former case is much more likely to happen than the latter, since nodes typically alternate work and sleep periods to save energy.

Moreover, the combination of leased subscriptions and broadcast communication remarkably simplifies the management of the subscription table, and drastically reduces the associated computational and memory overhead. In [9], to properly reconcile subscription information upon connectivity changes, we kept a different table for each value of ϕ, where each row contained the subscription filter and the link the subscription referred to. Here, instead, all we need is to store the subscription filter together with a timestamp used for managing leases. Differentiating according to ϕ is no longer needed, since subscriptions simply expire, and broadcast removes the need for information about links.

2.2. Routing Events

In [9], the effectiveness of event routing is controlled by means of the *event propagation threshold* τ, which is a fraction of the links available at a given node. For instance, $\tau = 0.5$ means that an event is always forwarded along half of the links available at each node. If subscription information is available, this is used first. If this deterministic information is not enough to fulfill the propagation threshold, the remaining links are selected at random among those the event has not been forwarded along. Clearly, higher values of τ increase not only delivery but also overhead. The simulations in [9] analyze the effect of this parameter in conjunction with the subscription horizon ϕ.

Nevertheless, in this paper we assume broadcast communication, therefore this strategy must be adapted slightly because there is no concept of link, actually leading to an even simpler strategy. If an event is received[3] for which a matching filter exists in the subscription table, the event is simply rebroadcast. On the other hand, if no matching subscription is found, the event is rebroadcast with a probability τ. The parameter τ, therefore, still limits the extent of propagation, but more indirectly than in [9], as it comes into play only when no deterministic information is available.

The effectiveness of our approach is clearly proportional to the number of forwarders F, i.e., the neighbors which receive and retransmit an event. Based on the procedure we described so far, in absence of deterministic information $F = \tau \cdot \eta$ holds, being η the number of neighbors. As a consequence, a small value of η (e.g., in sparse networks) must be compensated by increased values of τ, as we discuss in Section 4.

Moreover, while in [9] the event always got routed along the fraction of links mandated by τ, here instead we have a non-zero probability that none of the neighbors will rebroadcast the event. More precisely, in absence of deterministic information, if η is the average number of neighbors, the probability of stopping the propagation of the event is $(1-\tau)^{\eta}$. If no subscriber is in the immediate vicinity of the event publisher and τ is small, there is a significant possibility that event propagation immediately stops. To ensure that a reasonable amount of event messages are injected into the network, we mark event messages with a flag stating whether they have been just published or instead they already travelled through the network. In the first case, the receiver behaves as if $\tau = 1$ and rebroadcasts the event in any case. This mechanism guarantees that at least η copies of the event message are injected in the network and propagate independently.

2.3. Dealing with Collisions

Wireless broadcast is subject to packet collisions, which occur when two or more nodes in the same area send data at the same time. Since in our approach the propagation of subscriptions and events both rely on wireless broadcast, it becomes crucial to reduce the impact of collisions by avoiding wasting precious energy on useless retransmissions.

TinyOS [14] adopts a very simple scheme to recover from collisions where, after a broadcast message has been sent, the sender waits for an acknowledgment from at least one of its neighbors. If none is received before the associated timeout expires, the message is resent. The evident weakness of this solution is that it does not take into account the actual number of neighbors. If only one neighbor received and acknowledged successfully the message, the transmission is assumed successful, regardless of the possibly many nodes that did not receive the message. Moreover, it does not try to limit in any way the number of collisions. More sophisticated MAC protocols has been proposed in literature [18] but none is currently supported by the Crossbow MICA2 [1], our target platform.

---

2 Optimizations are possible, e.g., to broadcast the subscription hash, and transmit the entire one only if missing on the receiving node.

3 Clearly, events that have already been processed and that are received again because of routing loops are easily discarded based on their identifier.

Therefore, we conceived a simple yet effective solution that decreases significantly the number of collisions, without requiring any synchronization among nodes. The idea can be regarded as a sort of simplified TDMA protocol where each node, upon startup, sets a timer whose value is a global configuration parameter. Sending messages (i.e., subscriptions and events) takes place only upon timer expiration, while receiving is in principle always enabled. Since each node in the network bootstraps at a different time, it is highly unlikely that two nodes in range of each other end up with synchronized timers. The simulations in Section 4 show that this trivial idea goes a long way in drastically reducing the amount of collisions.

2.4. Avoiding Unnecessary Propagation

Without a way to limit forwarding, an event propagates a long way—i.e., until it reaches a node that already received it, at which point it gets dropped. This unconstrained propagation is likely to generate unnecessary overhead. In [9] we addressed the problem by setting a time-to-live (TTL) on each event, incremented at each hop. However, our simulations showed that this solution is much less effective with broadcast propagation. In fact, even when an event travels for a small number of hops, the number of nodes it reaches is great, and therefore the impact of TTL is limited.

To address this issue, we modified slightly the retransmission strategy described in Section 2.3. Let us assume a node A waiting to broadcast an event e hears one of its neighbors, say B, transmitting e before A's timer expires. If the set of A's neighbors partially overlaps with B's neighbors, it is likely that most of A's neighbors receive the event from B's transmission, therefore making A's broadcast largely useless. Some of A's neighbors may not hear about e from B but, given the epidemic nature of our algorithm, they are very likely to get it through other routes.

Based on this observation, in our approach (which we called *delay-drop*) we would simply let A safely remove e from its transmission queue. In doing this, not only we limit propagation—our initial rationale for this modification—but also reduce communication and therefore save battery power. A downside of this approach is a potentially higher latency, as the event may go through longer routes before reaching its recipients. Nevertheless, in principle this delay-drop mechanism could be only one of many alternatives specified at the application or middleware layer, therefore enabling to tradeoff latency for overhead as needed.

3. Implementation

We implemented our approach for the Crossbow MICA2 [1] platform, using the *NesC* [11] language provided by TinyOS [14]. A TinyOS application is composed of *modules*, containing the actual code, and *configurations*, which are essentially module containers (components) describing how modules are wired together, and exporting interfaces that provides access to the overall component functionality. An interface contains function signatures, divided in *commands* (implemented by the interface provider) and *events* (implemented by the interface user).

```
configuration MHopRoutePubSub {
 provides {
  interface StdControl;
  interface Receive[uint8_t id];
  interface Send as SendSub[uint8_t id];
  interface Send as SendUnsub[uint8_t id];
  interface Send as SendPub[uint8_t id];
 }
 uses {
  interface ReceiveMsg as ReceiveMsgPub[uint8_t id];
  interface ReceiveMsg as ReceiveMsgSub[uint8_t id];
 }
}
implementation {
 components
  MHopRoutePubSubM,
  GenericCommPromiscuous as Comm,
  QueuedSend, TimerC, RandomLFSR;

 SendSub = MHopRoutePubSubM;
 SendUnsub = MHopRoutePubSubM;
 SendPub = MHopRoutePubSubM;
 Receive = MHopRoutePubSubM;
 StdControl = MHopRoutePubSubM;
 ReceiveMsgSub = MHopRoutePubSubM;
 ReceiveMsgPub = MHopRoutePubSubM;
 MHopRoutePubSubM.SubControl -> QueuedSend.StdControl;
 MHopRoutePubSubM.CommStdControl -> Comm;
 MHopRoutePubSubM.CommControl -> Comm;
 MHopRoutePubSubM.Random -> RandomLFSR;
 MHopRoutePubSubM.SendMsg -> QueuedSend.SendMsg;
 MHopRoutePubSubM.Timer -> TimerC.Timer[unique("Timer")];
}
```

Figure 1. NesC configuration for `MHopRoutePubSub`.

Architecture. Our implementation essentially provides a replacement of the standard TinyOS routing component, `MultiHopRouter`. The *nesC* configuration of the new module, called `MHopRoutePubSub`, is shown in Figure 1.

The first two blocks of the configuration define the interfaces provided and used by this component. The commands `SendPub`, `SendSub`, and `SendUnsub` are instances of the built-in generic `Send` interface defined by TinyOS, and deal with sending an event, a subscription, and an unsubscription, respectively. By “remapping” these interfaces on `Send` we are able to reuse a significant part of the lower-level code dealing directly with communication. `Receive` is also a standard TinyOS interface, and provides a way for the routing component to signal the application whenever a matching publish-subscribe event has been received. The `ReceiveMsg` interface, instead, is provided by the underlying communication component, and is used to signal the

```
typedef struct MultiHopMsgSub {
 uint16_t srcaddr; //source address
 uint8_t msgid;    //message identifier
 uint8_t subject;  //subject identifier
 uint8_t hopcount; //subscription hopcount
 uint8_t lease;    //subscription lease
} __attribute__ ((packed)) TOS_MHopMsgSub;

typedef struct MultiHopMsgPub {
 uint16_t srcaddr; //source address
 uint8_t msgid;    //message identifier
 uint8_t subject;  //subject identifier
 uint16_t data;    //event data
} __attribute__ ((packed)) TOS_MHopMsgPub;
```

Figure 2. Subscription and event messages.

routing component that a new network message has arrived. As in the case of `Send`, we "remap" this (TinyOS event) interface onto two different ones: `ReceiveMsgPub` and `ReceiveMsgSub`. Finally, `StdControl` is a common interface used to initialize and start all TinyOS modules.

The last block of the configuration specifies the list of modules used by this one, and how their interfaces are wired together. The main component is `MHopRoutePubSubM` which implements all the interfaces provided by `MHopRoutePubSub`. The others are TinyOS built-in modules: `GenericCommPromiscuous` and `QueuedSend` support message communication, `TimerC` provides the timer functionality necessary for leases and communication, and `RandomLFSR` provides the ability to generate random numbers.

Message structure. We defined two message types, one for subscriptions and another for events, shown in Figure 2. They both include the message source and a unique message identifier, which together enable duplicate detection. Also, our current implementation is subject-based, and therefore both messages include a subject identifier. An extension to content-based is straightforward. Besides these common fields, each subscription message also includes a `hopcount` field, which is initialized with the chosen value of the horizon ϕ and decremented at each hop, and a `lease` field, which contains the value in seconds during which a subscription is considered valid. Instead, event message contains a `data` field.

Handling subscriptions and events. Whenever the application issues a subscription, the corresponding subject is stored in a local subscription table. Moreover, a subscription message is broadcast to all the neighbors, with the `hopcount` initialized to ϕ. Subscriptions are kept alive by using a timer. When it fires, a new subscription message is sent for each subject in the local subscription table. An unsubscription simply consists of removing the corresponding subject from the local subscription table.

Non-local subscriptions are managed in a different subscription table. When a subscription message for a given subject is received, it is inserted in the table, possibly overwriting obsolete information for that subject with the new one containing a more recent lease. Moreover, if the `hopcount` is not zero, the subscription is enqueued, waiting to be rebroadcast according to the strategy we discussed in Section 2.3. Periodically, subscriptions whose lease expired are removed.

To handle events, `MHopRoutePubSubM` maintains a list of those most recently received. When an event message is received, this list is checked to see whether the event is a duplicate. In this case, the message is simply dropped. Otherwise, it is first inserted in the list, and then its subject is checked against the local subscriptions, to determine whether its receipt must be signaled to the application through the `Receive` interface. Then, it is checked against the non-local subscription table. If a subscription is found, the event message is inserted in the sending queue. Otherwise, a random number is drawn and, according to τ, either the event message is inserted in the sending queue or it is simply dropped.

Parameter	Value
Network Size	$N = 200$
Number of Neighbors	$\eta = 5$
Percentage of Receivers	$\rho = 10\%$
Publish Rate	2 event/s
Transmission Interval	1 s

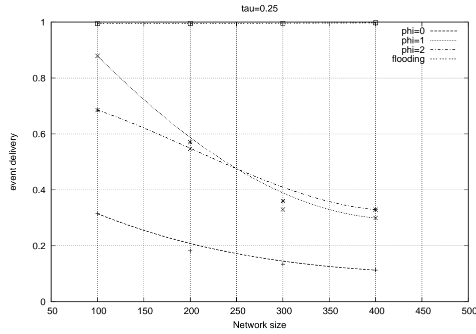
Table 1. Default values used in simulations.

4. Evaluation

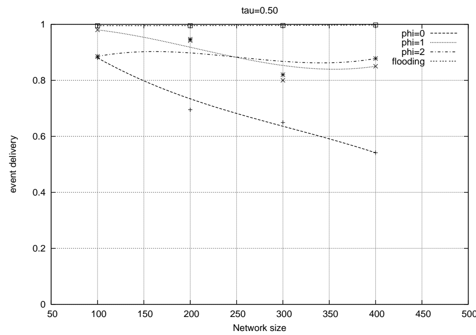
In this section we evaluate several aspects of our approach using TOSSIM [16], the simulation tool provided with TinyOS. TOSSIM emulates all the operating system layers and therefore works by reusing directly the code deployed on the motes, and described in the previous section.

Simulation Setting. Table 1 shows the most relevant parameters of our simulations, and their default values. Each simulation run lasted 60 simulated seconds, with an extra second devoted to "booting" the network, as performed automatically by TOSSIM. Transmission occurs by using our simple delay technique to avoid collisions. The impact of this technique, as well as of its delay-drop variant, is analyzed later in this section.

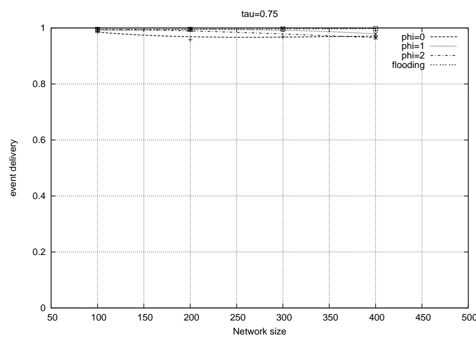
For each run we plot the event delivery (i.e., the ratio between the events expected to be received and the events actually received) and the overhead (i.e., the collective number of sent messages, including both events and subscriptions). To focus on these two performance metrics and reduce further bias, we ran our simulations with a stable set of subscriptions (i.e., no refresh needed) and a stable network connectivity (apart from the changes induced by duty cy-



(a) $\tau = 0.25$

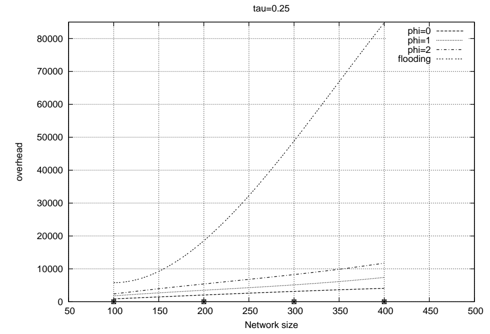


(b) $\tau = 0.5$

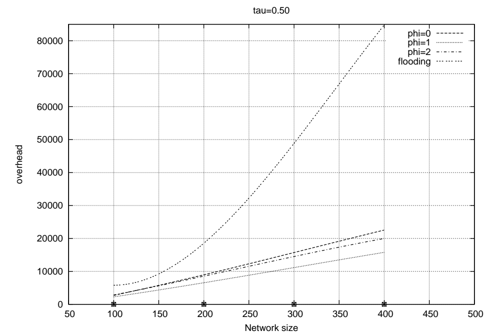


(c) $\tau = 0.75$

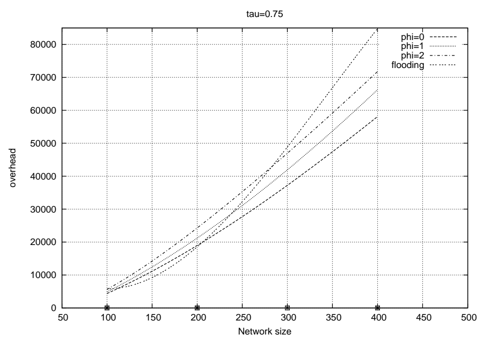
Figure 3. Delivery vs. network size.



(a) $\tau = 0.25$



(b) $\tau = 0.5$



(c) $\tau = 0.75$

Figure 4. Overhead vs. network size.

cle). Moreover, we analyze the behavior of our algorithms with different combination of τ and ϕ to estimate their impact. Our upper and lower bounds are *flooding* ($\tau = 1$) and a purely probabilistic approach ($\phi = 0$). Flooding delivers all the events but with very high overhead, while a fully probabilistic approach exhibits low overhead but at the cost of poor event delivery.

Network Size. The first parameter we analyze is the size of the network, which we ranged from 100 to 400. To maintain a steady publishing load and receiver density, we increased them proportionally by ranging the former from 1 to 4 evt/s, and keeping the latter at 10% (yielding from 10 to 40 receivers).

The results depicted[4] in Figure 3 confirm our expectations, showing that event delivery is only marginally dependent from the network size—at least for $\tau = 0.5$ and $\tau = 0.75$. This is not surprising, since the probabilistic component of our approach tends to distribute the load equally on each node and, therefore, the more the network grows (and the more receivers need to be reached), the more nodes participate in delivering the events. Notably, in some cases event delivery is even increased as more routes become available. On the other hand, as shown in Figure 4 the

4 We use Bezier interpolation to better evidence the trends.

overhead increases too, since the number of receivers and the publishing load augments linearly, i.e., there are more events to deliver to more recipients. Nevertheless, the two increments share the same trends, that is, no additional overhead is introduced by the size. This, again, stems from the fact the the effort imposed on each node by our algorithm is constant.

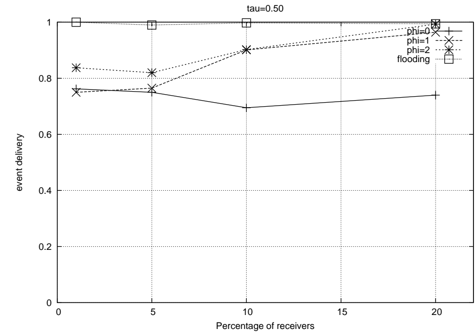
As the charts show, τ is tightly related to event delivery, since it controls the degree of propagation in the system. With values close to 1 (see Figure 3(c)), the system is able to improve event delivery up to 100% with the downside of an increase in network traffic (Figure 4(c)). In the extreme case of $\tau = 1$ (*flooding*) no event gets lost, but the network becomes overwhelmed by messages, since each node rebroadcasts all the events. Besides, with high values of τ collisions may drastically grow, thus hampering delivery. Therefore, the right value for τ is a tradeoff among delivery, overhead, and collisions.

On the other hand, the reason of the low performance achieved by using $\tau = 0.25$ lies in the fact that, as discussed in Section 2.2, the probability that no neighbor broadcasts an event is $(1-\tau)^{\eta} = 0.75^5 = 0.23$, i.e., one in four events are dropped by all neighbors. Figure 4(a) reflects this, by showing that the overhead is less than 20% of the flooding one.

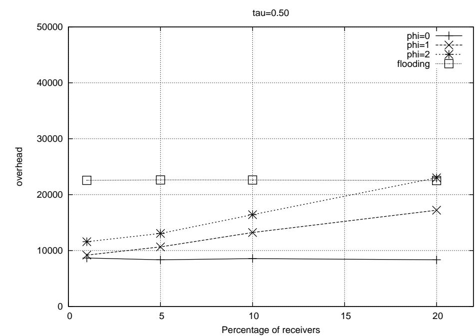
As for ϕ, it is interesting to see that $\phi = 1$ and $\phi = 2$ exhibit a different behavior. When $N = 100$, $\phi = 2$ performs worse than $\phi = 1$, most likely due to the fact that the smaller size increases the likelihood of creating loops. As N increases, however, the additional deterministic information provided by $\phi = 2$ becomes precious in steering events towards the receivers in a sparser network.

Finally, the comparison with flooding is also worth commenting. Indeed, the delivery with τ equal to 0.5 and 0.75 is essentially comparable, but overhead is sensibly lower. This is particularly evident for $\tau = 0.5$, which in this scenario represents the best tradeoff for costs and performance, being able to deliver about the 90% of events with about 25% of the overhead introduced by flooding.

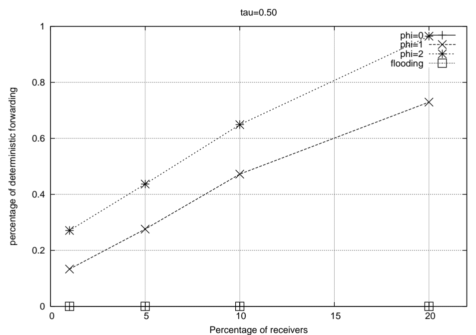
Number of Receivers. Another interesting view on our approach is the impact of ρ, the percentage of receivers. As shown in Figure 5, delivery with $\phi = 0$ is nearly unaffected by ρ and is about constant despite the increasing receivers. This is reasonable, since purely probabilistic routing makes essentially "blind" decisions, regardless of the presence of receivers. Conversely, with $\phi > 0$, delivery improves significantly with the number of receivers, as more deterministic information is available to each host. Figure 5(c) shows that indeed this information is increasingly exploited to steer events towards receivers as ρ increases. Also, it shows that $\rho = 20\%$ of receivers is enough to obtain, when $\phi = 2$, a routing that is basically entirely deterministic.



(a) Delivery.



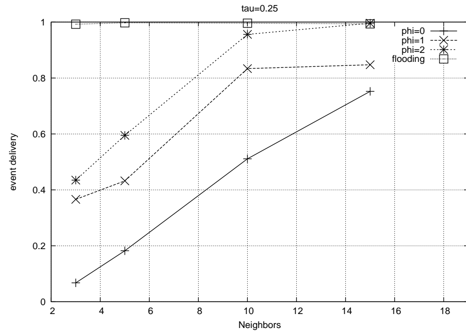
(b) Overhead.



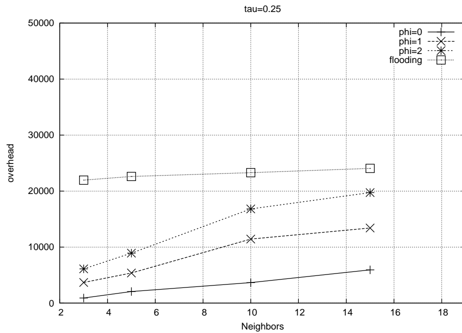
(c) Percentage of deterministic hops.

Figure 5. Number of receivers ($\tau = 0.5$).

Number of Neighbors. Another key factor that greatly impacts the performance of our approach is the network density, defined by the average number η of neighbors for each node. Not surprisingly, our approach performs worse in a sparse network, as fewer nodes participate in the event routing. Figure 6 analyzes the performance by ranging from $\eta = 3$ to $\eta = 15$, for $\tau = 0.25$. This value of τ is particularly interesting, since in Figure 3 it led to the worst performance. Instead, Figure 6(a) show how the increase in η boosts performance remarkably. The bottomline is represented by the purely probabilistic approach, which experiences a linear increase in delivery. The reason is that, as

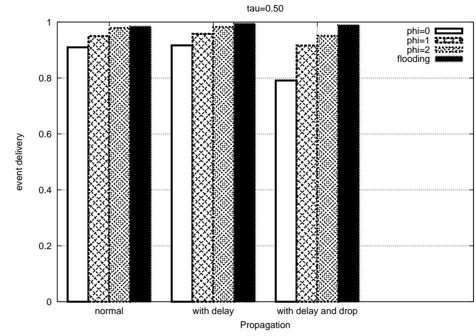


(a) Delivery.

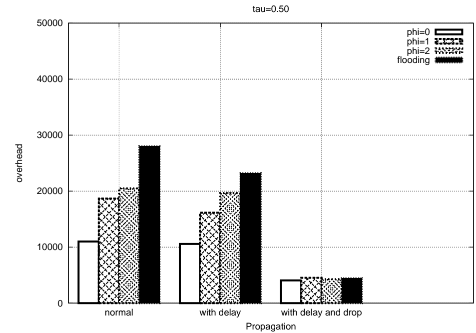


(b) Overhead.

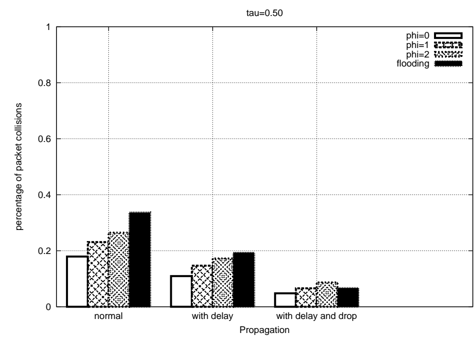
Figure 6. Number of neighbors ($\tau = 0.25$).



(a) Delivery.



(b) Overhead.



(c) Number of collisions.

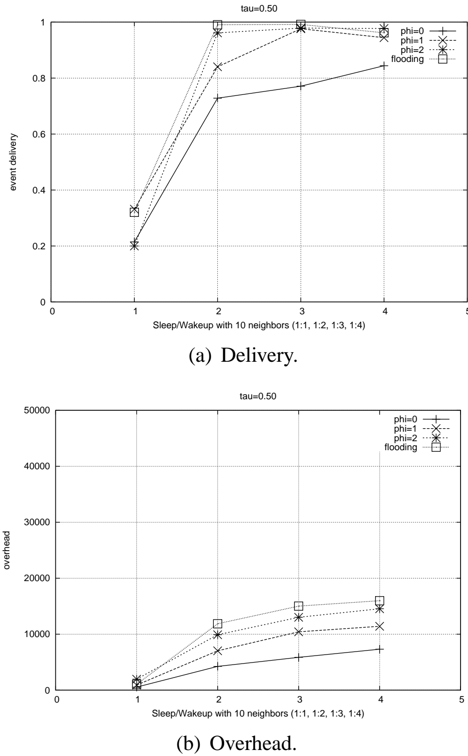
Figure 7. Collisions and delay-drop ($\tau = 0.5$, $\eta = 10$).

stated earlier, delivery is directly proportional to the number of forwarders F, which in turn depends directly on η and τ. Therefore, low values of τ are sufficient in a dense network. Moreover, the curves with $\phi > 0$ converge much faster to a 100% delivery, showing that deterministic information definitely improves delivery. At the same time, Figure 6(b) shows how this is achieved by keeping overhead reasonably low.

The effect is still observable, although less marked, with greater values of τ, not reported here. In this case, even with a sparse network the number of forwarders F is sufficient to achieve a satisfactory event delivery. Indeed, we verified that the performance with $\tau = 0.25$ and $\eta = 10$ is about the same of the one obtained with $\tau = 0.5$ and $\eta = 5$. Given this analysis, it should be noted how our choice of $\eta = 5$ as the default value in our simulations is rather conservative.

Collisions and Rebroadcast. In Section 2.3 and 2.4 we described two simple techniques for, respectively, reducing collisions and avoiding useless rebroadcasts.

The effect of these techniques on the system is shown in Figure 7 for $\tau = 0.5$ and $\eta = 10$. Figure 7(a) shows that the delivery is largely unaffected, with a small decrease in the case of delay-drop. On the other hand, Figure 7(c) shows that our simple mechanism for avoiding collisions is very effective, since it more than halves the number of collisions. The delay-drop mechanism does not improve much in terms of collisions. Instead, by avoiding useless rebroadcasts, this latter technique drastically reduces overhead, as shown in Figure 7(b). Although we do not have simulations linking directly these results to the power consumption, it is evident how the combination of these two simple techniques not only improves the performance of our approach, but also yields remarkable savings in communication, therefore enabling a longer life of the overall sensor network.



(a) Delivery.

(b) Overhead.

Figure 8. Sleeping nodes ($\tau = 0.5$, $\eta = 10$).

Duty Cycle. A prominent feature of our approach is the resilience to changes in the underlying topology and connectivity. Most approaches for content dissemination and group communication for sensor networks rely on exact routes that must be recalculated each time the topology is modified. However, this is an important limitation, since sensors are often supposed to regularly switch from active to sleeping, to preserve battery and extend the system lifetime. Therefore, unless some kind of synchronization is in place, routes become invalid and must be recomputed, with consequent overhead. Conversely, our approach does not make any assumption on the underlying topology, as it "explores" it semi-probabilistically. Therefore, it can tolerate sleeping nodes (or even crashed, or moving) nodes, without any particular trick.

In the simulations in Figure 8, we used a simple model where each node is active for a period T_a, followed by a sleeping period T_s. All nodes are initially active: after a random time (which temporally scatters them) they are regularly switched off and reactivated after T_s. To obtain meaningful results, sleeping nodes are not considered in the event delivery, which is then computed by taking into account only the active subscribers. Also, since the temporal scattering among nodes is completely random, it may happen that under certain combination of T_a, T_s and η, the network becomes not connected. Then, a delivery of 100% is not meaningful because, if no path exists among two nodes, there is no way to correctly deliver the event. Consequently, our upper bound is represented by the delivery of flooding.

By comparing Figure 8(a) at $T_a = 3T_s$ and Figure 6(a), it can be noted how the event delivery is quite similar, although a significant fraction of the nodes (about 20%) is unable to receive or forward events[5]. If more nodes are sleeping at the same time, delivery falls to 60%, since the number of forwarders is too low. However, increasing τ is sufficient to achieve a high delivery while maintaining a reasonable overhead.

These results are not surprising, since what we stated earlier about density holds here as well. Indeed, the effect of sleeping nodes is to reduce the density, expressed in terms of the number η of neighbors. Therefore, since our algorithm tolerates low densities up to a given extent, it is resilient to sleeping nodes as well. The validity of this statement is shown by observing that 50% of the nodes sleeping in a network with $\eta = 10$ is roughly equivalent to a network where all nodes are active and $\eta = 5$.

5. Related Work

Although sensor networks have been studied for some years, research has focused only recently on the development of reusable middleware platforms as opposed to all-in-one solutions. As a consequence, there are several promising works (e.g., [13,20,23]), many of which, however, focus on architecture design or run-time language support rather than routing issues.

A more meaningful comparison is with works addressing multicast or group communication in sensor networks. Unfortunately, the target scenario pursued by these middleware is mostly characterized by a set of sensors spread in the environment that cooperate to deliver the sensed data to a fixed node acting as base station or, alternatively, to enable communication from the base station towards all the sensors (e.g to perform a query or to force a network reprogramming). This hampers these solutions to be successfully exploited in the situations where sensors require data from other nodes to execute their task (as in the aforementioned WSAN networks) or to perform in-network processing and aggregation.

By and large, two main approaches have been investigated in literature: *flooding* [15] and *tree-based routing* [17, 21]. Flooding is a simple approach that offers the lowest control overhead at the expense of generating very high data

5 In most scenarios found in literature, sensor nodes sleep for most time and switch on only for a short amount of time. However, in our scenario, sensor nodes are essential not only to acquire data from the environment but also to participate in their propagation. Hence it seems reasonable that the ratio $\frac{T_a}{T_s}$ is greater than (or at least equal to) 1.

traffic in a wireless environment as we have shown in Section 4. The tree-based approach, on the other hand, generates minimal data traffic in the network, but tree maintenance and updates require many control messages and, more importantly, a stable network. A more refined algorithm [6] spreads nodes' interests across the whole network to create a reverse path from a publisher to receivers. However, again, no details are provided about how to deal with a dynamic network, as in the case of mobile or sleeping sensors, and failures.

The possibility of temporarily switching off nodes is particularly amenable in sensor networks as the battery is not easily replaceable. At the same time, however, the network must maintain its functionality through a connected subnetwork, i.e., it should be able to correctly deliver events despite the lack of some nodes. Some works [7, 22] address this issue by introducing synchronization of the sleeping patterns to minimize the energy spent without affecting network connectivity. The weakness of this solution, however, is that other kinds of topological reconfiguration (e.g., mobility or failures) are not tolerated. In these cases, the (expensive) synchronization procedure must be restarted, with added overhead. Conversely, our approach does not require any synchronization protocol and yet tolerates arbitrary reconfigurations.

6. Conclusions and Future Work

In this paper we proposed a routing approach enabling publish-subscribe on sensor networks. The routing strategy is semi-probabilistic, in that it relies on deterministic subscription information being disseminated close to the subscriber and, where this is absent, resorts to random rebroadcast. The approach described in this paper is inspired by our earlier work [9], which we adapted and extended here to better suit the peculiarity of the wireless sensor network environment. The results show that our approach provides good performance in terms of high delivery and low overhead, and is resilient to changes in connectivity, therefore making it amenable to our target deployment scenario.

References

[1] Crossbow Technology Inc. `http://www.xbow.com`.

[2] Sensor networks applications. *Special Issue of IEEE Computer*, 37(8):41–78, 2004.

[3] Wireless sensor networks. *Special issue of Communications of the ACM*, 47(6), June 2004.

[4] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal (Elsevier)*, 2(4):351–367, October 2004.

[5] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and E. E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[6] D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *Proc. of the 1st International Workshop on Wireless Sensor Networks and Applications*, pages 22–31, 2002.

[7] J. Carle and D. Simplot. Energy-efficient area monitoring for sensor networks. *IEEE Computer*, 37(2):40–46, 2004.

[8] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS04)*, pages 552–561. IEEE Computer Society Press, march 2004.

[9] P. Costa and G. Picco. Semi-probabilistic content-based publish-subscribe. In *Proc. of the 25th IEEE Int. Conf. on Distributed Computing Systems (ICDCS05)*, Columbus (Ohio, USA), June, 7-10 2005. To appear.

[10] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco. TINYLIME: Bridging Mobile and Sensor Networks through Middleware. In *Proc. of the 3rd IEEE Int. Conf. on Pervasive Computing and Communications (PerCom 2005)*, pages 61–72, Kauai Island (Hawaii, USA), Mar. 2005.

[11] D.Gay et al. The NesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 1–11. ACM Press, 2003.

[12] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[13] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[14] J. Hill et al. System architecture directions for networked sensors. In *Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104. ACM Press, 2000.

[15] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of the 6th International Conference on Mobile computing and networking (MOBICOM)*, pages 56–67, Boston, MA USA, 2000.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of the 1st Int. Conf. on Embedded Networked Sensor Systems (SenSys'03)*, pages 126–137. ACM Press, 2003.

[17] S. Madden, M. Franklin, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of SIGMOD 2003*, 2003.

[18] P. Naik and K. M. Sivalingam. *Wireless Sensor Networks*, chapter A survey of MAC protocols for sensor networks. Kluwer Academic Publishers, 2004.

[19] G. Picco, G. Cugola, and A. Murphy. Efficient Content-Based Event Dispatching in Presence of Topological Reconfiguration. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems (ICDCS03)*, pages 234–243. ACM Press, May 2003.

[20] E. Souto, G. Guimares, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz. A message-oriented middleware for sensor

networks. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 127–134, New York, NY, USA, 2004. ACM Press.

[21] C. Srisathapornphat, C. Jaikaeo, and C.-C. Shen. Sensor information networking architecture. In *Proceedings of the 2000 International Workshop on Parallel Processing*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.

[22] D. Tian and N. D. Georganas. A coverage-preserving node scheduling scheme for large wireless sensor networks. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, pages 32–41, 2002.

[23] Y. Yu, B. Krishnamachari, and V. K. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18(1):15–21, 2004.