P&P Protocol: Local Coordination of Mobile Sensors for Self-deployment

Novella Bartolini, A. Massini, S. Silvestri
Department of Computer Science
University of Rome "Sapienza", Italy
{bartolini, massini, simone.silvestri}@di.uniroma1.it

ABSTRACT

The use of mobile sensors is of great relevance for a number of strategic applications devoted to monitoring critical areas where sensors can not be deployed manually. In these networks, each sensor adapts its position on the basis of a local evaluation of the coverage efficiency, thus permitting an autonomous deployment. Several algorithms have been proposed to deploy mobile sensors over the area of interest. The applicability of these approaches largely depends on a proper formalization of rigorous rules to coordinate sensor movements, solve local conflicts and manage possible failures of communications and devices. In this paper we introduce P&P, a communication protocol that permits a correct and efficient coordination of sensor movements in agreement with the PUSH&PULL algorithm. We deeply investigate and solve the problems that may occur when coordinating asynchronous local decisions in the presence of an unreliable transmission medium and possibly faulty devices such as in the typical working scenario of mobile sensor networks. Simulation results show the performance of our protocol under a range of operative settings, including conflict situations, irregularly shaped target areas, and node failures.

Categories and Subject Descriptors: C.2.2 [Computer Communication Networks]: [Network Protocols]

General Terms: Algorithms, Performance.

Keywords: Wireless sensor networks, mobile sensors deployment, distributed coordination protocol.

1. INTRODUCTION

The research in the field of mobile wireless sensor networks is motivated by the need to monitor critical scenarios such as wild fires, disaster areas, toxic regions or battlefields, where static sensor deployment cannot be performed manually.

In these typical working situations, sensors may be dropped from an aircraft or sent from a safe location. In these cases the initial deployment over the Area of Interest (AoI) is neither complete nor uniform as would be necessary to enhance the sensing capabilities and extend the lifetime of the network. Mobile sensors can dynamically adjust their position to improve coverage with respect to their initial deployment. Sensor movements should therefore be coordinated according to a distributed deployment algorithm.

Out of the solutions proposed in the literature so far for mobile sensor deployment, those described in [13, 7, 5, 10] are based on the virtual force approach which models the interactions among sensors as a combination of attractive and repulsive forces. Other approaches are inspired by the physics of fluids and gases such as [9] and [8]. Another methodology is based on the construction of Voronoi diagrams [12, 11]. According to this proposal, each sensor iteratively calculates its own Voronoi polygon, determines the existence of coverage holes and moves to a better position if necessary. The solutions proposed in [3] and [2] provide instead density driven actions to uniformly distribute sensors according to a regular grid pattern.

The applicability of these deployment algorithms largely depends on the proper formalization of rigorous rules to coordinate sensor movements, solve local conflicts and manage possible failures of communications and devices. Previous proposals only focus on the design of distributed algorithms for the adaptive deployment of mobile sensors, aiming at covering the area of interest according to given efficiency objectives, in particular coverage completeness and uniformity and low energy consumption. Seldom do previous works enter the details of the communication protocol necessary to enable the application of the proposed algorithms.

The main contribution of this paper is a communication protocol that defines the rules to deploy mobile sensors according to the PUSH & PULL algorithm proposed in [2]. This algorithm is based on the autonomic computing paradigm. It completely delegates to the single sensors every decision regarding movements and action coordination. This way self-organization emerges without the need of external coordination or human intervention as the sensors adapt their position on the basis of their local view of the surrounding scenario.

Given the absence of a centralized coordination unit, and the lack of synchronization, sensors have a primary role in the realization of the algorithm actions. Therefore, the design of the related coordination protocol is particularly challenging.

Indeed, under the execution of the PUSH & PULL algorithm, several types of conflicts may occur as several sensors often compete to cover the same position. Sensors should be capable to solve such conflicts by means of only local inter-

MSWiM'09, October 26–29, 2009, Tenerife, Canary Islands, Spain.

actions. We deeply investigate and solve the problems that may occur when coordinating asynchronous local decisions in the presence of an unreliable transmission medium and possibly faulty devices that characterizes the typical working scenario of mobile sensor networks.

The proposed protocol works in respect of the algorithm goals, permitting the realization of a complete and uniform stable coverage, with low energy consumption. Simulation results show the performance of our protocol under a range of operative settings, including conflict situations, irregularly shaped target areas, and node failures.

2. THE PUSH & PULL ALGORITHM

The purpose of PUSH & PULL is to let sensors form a hexagonal tiling that constitutes a complete coverage of the AoI and a connected network deployment. Notice that the hexagonal tiling corresponds to a triangular lattice arrangement, that is the one that guarantees at the same time network connectivity, optimal coverage extension and density, as discussed in [4]. The design of PUSH & PULL is based on the idea to make some sensors stick to the hexagonal grid points and let the others uniformly distribute over the whole AoI. According to the PUSH & PULL algorithm, sensors are involved in four basic activities executed in an interleaved manner: 1) **Snap**, described in Section 4, which makes the sensors move and stick to the grid points of the hexagonal tiling, 2) **Push**, described in Section 5, which allows the flow of non-snapped sensors towards low density areas, 3) **Pull**, described in Section 6, which attracts non-snapped sensors toward coverage holes, and 4) **Merge**, described in Section 7, which makes several grid portions merge into a unique regular hexagonal tiling. A fifth activity, **role exchange**, described in Subsection 5.3, is introduced to balance the energy consumption among the available sensors. Note that the P&P protocol we propose in this paper implements these activities without the need of global synchronization among sensor, as it will be explained in the next sections. More details on the activities at the basis of the PUSH & PULL algorithm can be found in [2]. For the sake of clearness, in Figure 1 we give an example of the protocol execution over a rectangular AoI. We refer to this figure throughout the paper to describe the main activities implemented by our protocol.

3. THE P&P PROTOCOL

The implementation of the PUSH & PULL algorithm requires the definition of a protocol for the local coordination of the sensor activities.

The coordination protocol provides the rules to solve contentions that may happen in several cases. For example, two or more snapped sensors can decide to issue a snap command to different sensors towards the same hexagon tile or the same low density hexagon can be selected by several snapped sensors as candidate for receiving redundant sensors. These contentions are solved by properly scheduling actions according to message time-stamps and by advertising related decisions as soon as they are made.

The P&P protocol is designed to minimize energy consumption entailing a small number of message exchanges, which is possible because the algorithm decisions are only based on a small amount of local information. Furthermore, we assume that P&P works over a communication protocol stack which handles possible transmission errors and message losses by means of timeout and retransmission mechanisms. Therefore the treatment of occasional message losses at the underlying protocol level implies the occurrence of delays in the corresponding messages at the P&P level that are dealt by P&P with proper timeout mechanisms.

Before we enter the details of the protocol P&P we introduce some definitions. V is a set of equal sensors endowed with location determination, boolean sensing and isotropic communication capabilities. Notice that location awareness (usually obtained by means of GPS devices) is only necessary in the case of sensor deployment over a specific target area. If sensors are to be deployed in an open environment, the assumption of location determination capability can be removed, as in other works in the area [6].

The deployment consists in realizing a hexagonal grid with side length $l_{\mathtt{h}}$ less or equal to the *sensing radius* $R_{\mathtt{s}}$. This setting guarantees both coverage and connectivity when the *transmission radius* $R_{\mathtt{tx}}$ is such that $R_{\mathtt{tx}} \geq \sqrt{3}R_{\mathtt{s}}$.

A sensor which is deployed at the center of a hexagonal tile is called *snapped*. $Hex(p)$ is the hexagonal region whose center is covered by the snapped sensor p. All the other sensors lying in $Hex(p)$ are called *slaves of* p and compose the set $S(p)$. All the sensors that are neither snapped nor slaves are called *free*. The set composed by the free sensors located in radio proximity to p and by its slaves is denoted by $L(p)$. The set $\mathrm{VP}(p)$ of vacant positions detected by sensor p contains the centers of hexagons adjacent to $Hex(p)$ that are not yet occupied by any snapped sensor.

Table 1 contains a summary of the message types used by protocol P&P.

4. P&P: SNAP ACTIVITY

In order to describe the snap activity, we need to distinguish three cases, according to the role of the involved sensor. Indeed the actions undertaken by the starter sensors, the already snapped sensors and the sensors being snapped, are substantially different.

4.1 Starter sensor behavior

At the beginning, any sensor p may give start to the creation of a tile portion by snapping itself to its present position in an instant of time $t_{\mathtt{start}}(p)$ randomly selected over a time interval of length $R_{\mathtt{tx}}/v$, where v is the sensor movement speed. If at the instant $t_{\mathtt{start}}(p)$, sensor p has not yet received any message, it elects its position as the center of the first hexagon and establish the orientation of its tile portion. At this point p executes the snap actions under the role of snapped sensor, as described in the following paragraph.

4.2 Snapped sensor behavior

4.2.1 Neighbor Discovery

A snapped sensor p broadcasts a `IAS` (I Am Snapped) message to perform a neighbor discovery. Such message contains the ID of the sender snapped sensor, its geographic coordinates and the timestamp of the starter action. All sensors located in radio proximity to p (with the exception of those slaves located in different hexagons) reply to its `IAS`, with a message containing role dependent information: the snapped sensors reply with an `InfoSnapped` message, while the slave and the free sensors reply with an `InfoSlave` and an `InfoFree` message respectively. All three types of replies

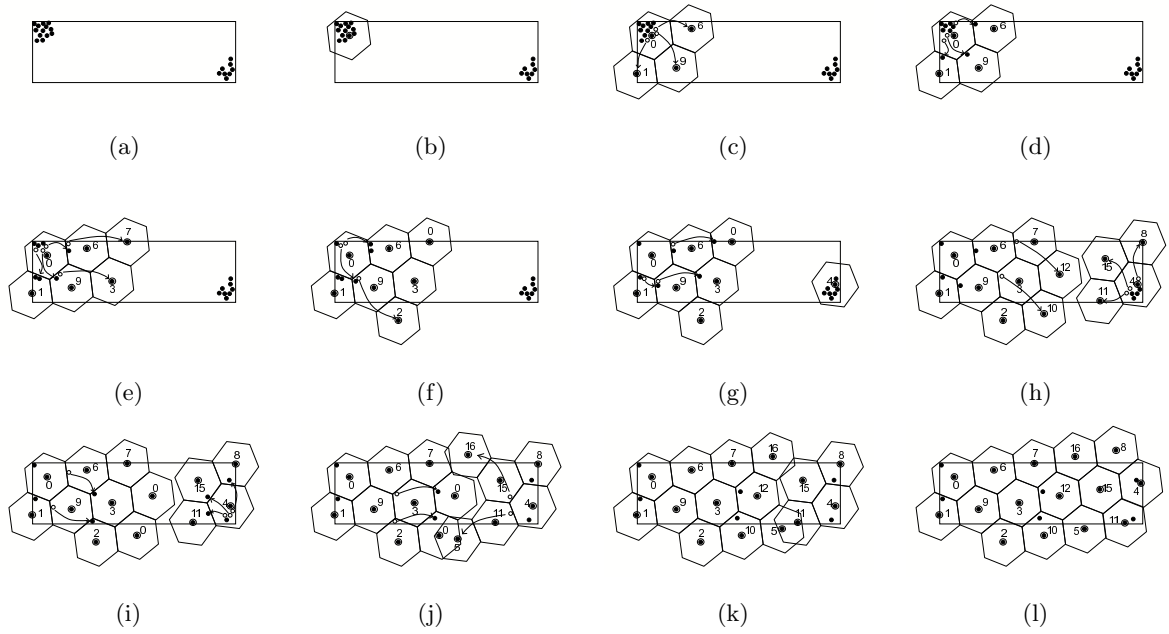


Figure 1: Algorithm execution: an example

contain the ID and geographic coordinates of the replying sensors. In addition, the `InfoSnapped` message includes also the *virtual cardinality* of the replying snapped sensors, that is the number of slave sensors located inside the hexagon of the sender as if all the movements were already concluded. The `InfoSlave` message includes the energy level of the replying slave sensors.

Thanks to the execution of the neighbor discovery phase, a snapped sensor p is informed regarding the presence of vacant positions, i.e. knows the composition of VP(p), and is able to build the set $L(p)$.

4.2.2 Snap into position

A snapped sensor p selects the closest sensor in $L(p)$ to each uncovered position and sends it a `SIP` (Snap Into Position) message. This message contains the target position of the correspondent snap action, and the ID of the selected sensor.

If a sensor receives a `SIP`, and is available to fill the vacant position, it replies with an `AckSIP` message. This message contains the ID of the sensor that received the `SIP`, necessary for p to discriminate among the several sensors to which it sent `SIP` messages. If a sensor receives a `SIP` when it is not available to fill the vacant position (e.g it has already been contacted by another sensor), it does not reply to the `SIP` message of p and lets the `AckSIP` timeout expire. This way p will be capable to select a new sensor to snap in such still vacant position.

After the transmission of the `SIP` messages and the reception of the related `AckSIP`, p updates its local information, i.e. the number of free sensors located within its transmission range and its virtual cardinality. This way it keeps into account the departure of some sensors from either its transmission range or its hexagon.

In order to update the information related to the snapped neighbors, p waits for the reception of the corresponding `IAS` messages, to be sure that position conflicts are solved (see 4.3.3). No messages are involved in this phase that consists in a mere calculation based on locally available information.

Let p be the sensor that is performing the snap action and let q be the one to which p sent a `SIP` message for the position x. Five cases may occur, described as follows.
1) Sensor p receives both the `AckSIP` and the `IAS` message from q. This means that the snap action performed by p was successful, therefore p can update the local information regarding the snapped neighbor q.
2) Sensor p receives the `AckSIP` from q acknowledging its availability to fill position x, but a conflict occurs solved in favor of another sensor r, which reaches position x before sensor q. Hence p receives an `AckSIP` from q and a `IAS` from r for the same position x. Thus p can update the local information regarding the snapped neighbor r.
3) Sensor p receives the `AckSIP` from q acknowledging its availability to fill position x, but a failure occurred and the `IAS` timeout expires. If p detects the availability of another sensor in $L(p)$ that can be snapped to position x, it retries the snap action. If such sensor is not available, p starts the pull action.
4) Sensor p does not receive the `AckSIP` from q, but receives a `IAS` message for position x from another sensor r, before the expiration of the `AckSIP` timeout. Sensor p can update the local information regarding the snapped neighbor r.
5) Sensor p does not receive the `AckSIP` from q nor the `IAS` from any other sensor within the `AckSIP` timeout. If p detects the availability of another sensor in $L(p)$ that can be snapped to position x, it retries the snap action. If such sensor is not available, p starts the pull action.

At the end of the snap activity, a snapped sensor p sends a `CardinalityInfo` message to its neighborhood. This message contains the ID and the virtual cardinality of p.

4.3 Behavior of the sensors being snapped

Message name	Message fields
IAS	ID, coordinates, starter timestamp
InfoSnapped	ID, coordinates, cardinality
InfoSlave	ID, coordinates, energy level
InfoFree	ID, coordinates
SIP	ID, receiver ID, target position coordinates
AckSIP	ID, receiver ID
ClaimPosition	ID, coordinates, timestamp
PositionTaken	ID, coordinates
InfoStopped	ID, coordinates
IAYS	ID, receiver ID
CardinalityInfo	ID, cardinality
Offer	ID, receiver ID, cardinality, transaction ID
AckOffer	ID, receiver ID
PositionTaken	ID, coordinates
AckInfoArrived	ID, receiver ID
MoveTo	ID, receiver ID, dest. coord., dest. snapped sensor ID, trans. ID
InfoArrived	ID, receiver ID, transaction ID, energy level
HoleInfo	ID, hop counter, order value, hole coordinates, timeout
Subst	ID, receiver ID, energy level
AckSubst	ID, receiver ID
SubstArrival	ID, receiver ID
ProfilePacket	ID, receiver ID, order value, priority queue, neighborhood info.
MoveToSubst	ID, receiver ID, order value, priority queue, neighborhood info.
Retirement	ID, hole coordinates

Table 1: Summary of the P&P messages

4.3.1 Sensor localization

A free sensor q which receives a `IAS` message, coming from a snapped sensor p, replies with either an `InfoFree` or an `InfoSlave` message depending on its position with respect to p. If q is located outside the hexagon of p, it remains in the free state and replies to p with an `InfoFree` message. If instead q is located inside the hexagon of p, it switches its state to slave and replies to p with an `InfoSlave` message. In both cases q becomes part of the set $L(p)$, that is the set of sensors that p can snap to its adjacent vacant positions. Notice that if q is a slave, there is only one snapped sensor p such that $q \in L(p)$, thus slaves belonging to already snapped sensors do not reply to the `IAS` message of p. If instead q is a free sensor, it may belong to several sets $L(\cdot)$, for different snapped sensors located in radio proximity from q itself.

4.3.2 Snap into position

Sensor q, be it free or slave, at a certain time, may receive a `SIP` message coming from a snapped sensor. Slaves reply only to `SIP` messages coming from their related snapped sensor, while free sensors only reply the first `SIP` message they receive and ignore subsequent ones.

After sending the `AckSIP` reply, sensor q travels towards the snapping destination until it reaches a distance d from it. Distance d is set small enough to guarantee the radio connectivity within the circular disk of radius d and the inclusion of such disk into the hexagonal tile. Therefore $d \leq \sqrt{3}l_{\mathrm{h}}/2$.

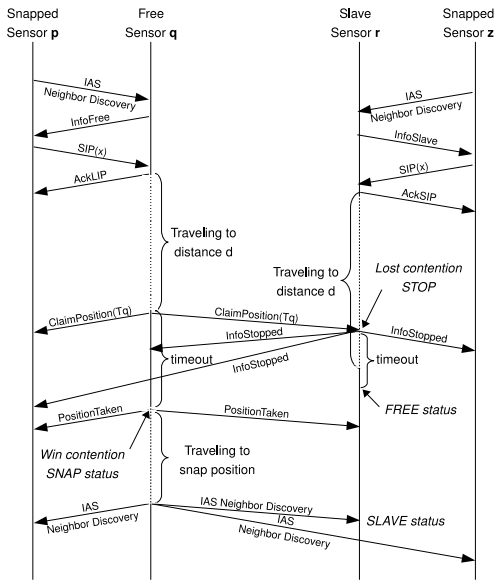


Figure 2: A typical scenario of snap position conflict

At this point sensor q stops and broadcasts a `ClaimPosition` message containing a timestamp and waits for the expiration of a timeout to evaluate if other sensors are trying to snap in the same position and in case to resolve the related contention. At the timeout expiration, if no conflicts occurred or if a conflict was solved in its favor, q switches its state to snapped, sends a `PositionTaken` message and proceeds towards the destination. After being successfully snapped, sensor q starts its own snap activity.

4.3.3 Resolution of snap position contention

Three events may occur when one or more sensors are engaged in a conflict with sensor q due to the contention for the same snap position:
1) sensor q receives a `ClaimPosition` or a `PositionTaken` before reaching distance d from the destination,
2) sensor q receives a `ClaimPosition` after the arrival at distance d from the destination and before the expiration of the related timeout,
3) sensor q receives a `PositionTaken` as a response to its `ClaimPosition`. This case may happen if q started travelling toward the destination when it was too far to perceive the previous `ClaimPosition` and `PositionTaken` messages.

In the first case, q stops moving and sends an `InfoStopped` message, to advertise its new position to the neighborhood, and starts a timeout. Snapped sensor receiving an `InfoStopped` message, verifies if the sender is inside its hexagon and in this case replies with a `IAYS` message (I Am Your Snapped), containing the sender and the receiver ID. If the stopped sensor receives a `IAYS` reply within the timeout, it sets its status to slave. Otherwise, if the timeout expires, it sets its status to free, not belonging to any hexagon.

In the second case, sensor q compares its timestamp with the one included in the `ClaimPosition` message. The sensor with lower timestamp wins the competition for the destination and proceeds its travel, sending a `PositionTaken` message, while the other sensor waits for the arrival of the

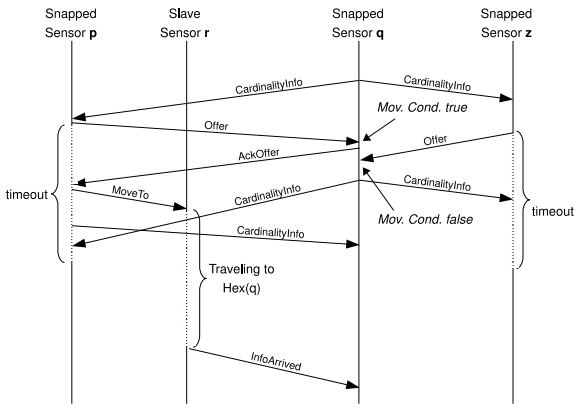


Figure 3: A typical scenario of the push activity

`IAS` message of the new snapped sensor to switch its status to slave.

In the third case, sensor q sets its state to slave of the newly snapped sensor. Notice that this timestamp based conflict resolution is designed to avoid redundant replies to `ClaimPosition` messages and does not require global synchronization.

Figure 2 shows a typical conflict resolution scenario, where two sensors r and q receive a `SIP` message for the same position x from two different snapped sensors. Both r and q start travelling towards the destination x. Sensor q reaches distance d from the destination before sensor r, and sends a `ClaimPosition` message, with its timestamp. Sensor r receives such message while travelling, and consequently stops because the contention for position x was won by sensor q. Sensor r sends an `InfoStopped` message to alert its neighborhood of its new position and starts a timeout. In the case depicted in Figure 2, r stops inside the hexagon centered in position x. For this reason, no snapped sensor replies to the `InfoStopped` message, thus after the timeout expiration, sensor r switches its status to free. After the expiration of the contention timeout, sensor q broadcasts a `Position-Taken` message and switches to the snap status while definitely travelling to position x. When q reaches position x, it starts a neighbor discovery by sending a `IAS` message, in consequence of which, r switches its status to slave.

In order to show an example of the snap activity execution, we refer to Figure 1. Figure 1 (a) and (b) show that, at the beginning of our example of P&P execution, only one sensor assumes the starter role. In figure 1 (c), this sensor snaps three of its slaves. In a second time, see Figures 1 (f) and (g), another node acts as starter and initiates the formation of a second grid portion by snapping three of its slaves as described in Figure 1 (h).

5. P&P: PUSH ACTIVITY

To describe the push activity we distinguish the behavior of snapped and slave sensors and illustrate the role exchange mechanism introduced to uniform the energy consumption.

5.1 Behavior of snapped sensors

5.1.1 Push proposal

As soon as a snapped sensor p terminates the snap activity, it sends a `CardinalityInfo` message to its neighborhood. Such message contains its ID and its virtual cardinality. Neighbor snapped sensors that receive this message update their information regarding sensor p and evaluate the opportunity to move slave sensors to its hexagon.

Even sensor p evaluates the opportunity to move some of its slaves to adjacent hexagons to uniform the distribution of redundant sensors. To this end, it uses its information regarding the neighbor snapped sensors, collected in the neighbor discovery phase. Sensor p looks for neighbor snapped sensors whose hexagons verify the Moving Condition [2] and have minimal cardinality. Among these, it selects the closest, to which it sends an `Offer` message containing its virtual cardinality, and an identifier of the current transaction, (transaction ID). If no sensor verifies the Moving Condition with p, sensor p waits for further events.

5.1.2 Push agreement

The snapped sensor q that receives an `Offer` message from p, verifies the validity of the Moving Condition as it could have more updated information than p. This way the responsibility of the slave movement is held by the receiver, thus ensuring that it only happens when the Moving Condition is actually valid. This is particularly important to guarantee the algorithm termination.

Two cases may occur: 1) q accepts the offer it received from p, or 2) q leaves the offer unreplied.

In the first case, q replies to p with an `AckOffer` message, containing only the recipient and sender ID. The sensor q updates its cardinality value, advertising the new value to its snapped neighbors, with a `CardinalityInfo` message. This way q can participate in further operations of distribution of redundant sensors with updated information. It also precludes other snapped sensors from sending unnecessary offers. When q accepts an offer, it starts a timeout identified by the transaction ID received in the `Offer` message. If q does not receive any message within the timeout, containing the related transaction ID, it decreases its cardinality and advertises this change with a new `CardinalityInfo` message. If, otherwise, sensor q receives an `InfoArrived` message related to the current transaction, it replies with an `AckInfoArrived`, containing its ID and the receiver ID. This way the protocol is robust to possible node failures during the push activity.

The sensor p selects a slave r to be pushed and sends it a `MoveTo` message containing the sender and receiver ID, the position and the ID of the destination snapped node (the sensor q), and the transaction ID. This selection is based on an energy saving criterion. The sensor p selects the slave sensor r that will remain with the highest energy after the completion of the entire movement.

5.2 Behavior of a slave sensor

The slave sensor r selected by the sensor p receives a `MoveTo` message and starts moving towards the hexagon of the sensor q. As soon as the sensor r crosses the boundary of the hexagon of q, it sends an `InfoArrived` message, stops moving and waits for the related `AckInfoArrived message`. The `InfoArrived` message contains the sender and receiver ID, the transaction ID, and the energy level of the sender. If

the AckInfoArrived message is not received within a timeout, sensor r assumes that sensor q is not there anymore. Thus it tries to snap in the snapping position of q, as if it would have received a SIP message for that position.

5.3 Role exchange

The PUSH & PULL algorithm provides that slaves and snapped sensors may occasionally exchange their roles in order to balance the energy consumption over the set of available sensors. Any time a slave r has to make a movement across a hexagon as a consequence of a push action, it sends a role exchange proposal consisting in a Subst message to the snapped sensor p of the hexagon it is traversing, and starts a substitution timeout. Subst messages contain the ID of sender and receiver, the energy level of the sender and the destination coordinates. The snapped sensor p uses the energy level value of r to decide if a role exchange may be of benefit in balancing the overall energy consumption between the two sensors. In this case, p replies with an AckSubst message.

If sensor r receives an AckSubst message within the substitution timeout, it travels toward the snap position held by sensor p, while p waits for the arrival of sensor r before starting to travel towards the destination initially targeted by r. Sensor r advertise its arrival to sensor p with a SubstArrival message containing the same fields of the AckSubst message. Sensor p replies to r with a ProfilePacket message that is necessary to enable a complete role exchange and starts travelling towards the destination.

If sensor r does not receive an AckSubst message within the substitution timeout, it continues its travel towards the destination.

Slave and snapped sensor substitutions may also occur at the beginning of the slave travel. In this case the substitution is started by the snapped sensor itself which already has all the available information to evaluate the opportunity to perform the role exchange. Under these circumstances, the snapped sensor p sends a MoveToSubst message containing the profile information necessary to perform the substitution. As soon as sensor r arrives in proximity to the snap position held by p, it sends the SubstArrival message described before, after which p starts travelling towards the destination.

5.4 An example

Figure 3 depicts a typical scenario of the push activity. The snapped sensor q broadcasts its virtual cardinality with a CardinalityInfo message. The snapped sensors p and z receive this message and verify the Moving Condition with the updated information received from q. As both p and z satisfy the condition, they send an Offer message to q. Notice that the Offer message always contains an updated value of the virtual cardinality of the sender. Since each node can offer at most one sensor at a time the virtual cardinality does not change until the offer timeout expires, or the receiver replies with an AckOffer message. Sensor q receives the Offer message from p before the one sent from sensor z. It verifies the validity of the Moving Condition with the updated virtual cardinality of p, received in the Offer message. As the Moving Condition is still satisfied, q replies with an AckOffer message, incrementing its virtual cardinality and broadcasting a CardinalityInfo message.

When node q receives the Offer message from z it verifies the Moving Condition again. Note that z sent this message on the basis of an old value of the virtual cardinality of q. Thus q finds that, as a consequence of the transaction just concluded with sensor p, the Moving Condition is unsatisfied with respect to sensor z, and consequently it does not reply to the offerer. Sensor z waits until the expiration of the offer timeout, after which it is able to be engaged in other push actions.

Sensor p receives an AckOffer message from q, thus it selects r within its slaves, and send it a MoveTo message. Sensor r moves towards the hexagon of q, and sends an InfoArrived message as soon as it arrives. Sensor p sends a CardinalityInfo message containing the decreased value of its virtual cardinality.

An example of the push activity execution can be found in Figure 1, where the values the virtual cardinality are shown. Figure 1 (d) shows that the snapped node 0 has some unsnapped nodes in its hexagon, and therefore starts the push activity towards its three adjacent hexagons. In Figure (e) the snapped nodes 6 and 9 perform the snap activity. Notice that the snapped sensor 1 does not perform any snap action as it does not have any hole around its hexagon. It also does not execute any push action as the Moving Condition is not satisfied. In the Figures (e) and (f), the snapped node 0 continues its push activity while the node 9 performs a snap of its slave. Notice the change in the *ord* value of the sensor number 7 in Figure (e) and (f). This change will be clear in the next section where we describe the pull activity in deeper details.

6. P&P: PULL ACTIVITY

In the present section we distinguish three possible roles of sensors involved in the pull activity.

A first role is the one of the sensor detecting a coverage hole in a neighbor location. This is the starter of the pull activity, which alters its *ord* value to enable push actions from nearby hexagons and sends related trigger notification messages.

The second role is the one of the neighbor snapped sensors which receive the trigger notification messages while not having available slaves to send. These sensors act as forwarder of the trigger messages in order to reach hexagons with redundant slaves that can be pushed to fill the holes.

The third role is performed by the snapped sensors which receive a trigger message when having available slaves to push. These sensors are informed of the changed *ord* value of the neighbor snapped sensors, and can contribute to fill the coverage holes by pushing the available slaves in the proper direction.

Notice that multiple trigger notification messages may reach the same sensor while performing any of the three listed roles. Such messages are inserted in a priority queue.

6.1 Sensors detecting coverage holes

A snapped sensor p, located in proximity of some vacant positions (i.e. $\mathrm{VP}(p) \neq \emptyset$), terminates the snap activity when no more sensors are available in $L(p)$. To give start to the pull activity, sensor p verifies if there is the possibility to attract sensors from its snapped neighbors. To this purpose, sensor p checks the validity of the Moving Condition with respect to all its snapped neighbors.

If p can not receive any sensor from its snapped neighbors, it starts the pull activity. To this purpose sensor p sets its

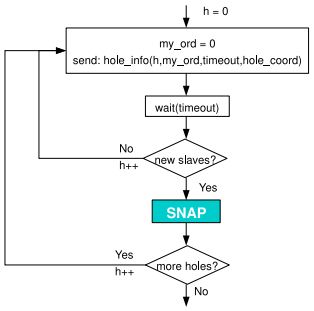


Figure 4: Behavior of a sensor detecting a hole

ord value to zero, and advertises this change by broadcasting a `HoleInfo` message containing its ID, a hop counter h, its updated *ord* value, the vacant position coordinates, and a timeout t_{out} which depends on the value of h (notice that this information is introduced to increase the algorithm efficiency). By modifying its *ord* value, sensor p alters the current situation with respect to the Moving Condition, enabling a new push activity from neighbor hexagons.

The hop counter h represents the forwarding horizon of the `HoleInfo` message, that is the distance to be traversed by this message, expressed in number of hexagons. Initially h is set to zero, thus the snapped sensors receiving a `HoleInfo` message only update their information about the sender *ord* value and do not forward this message. If no new slave reaches $Hex(p)$ within the given timeout t_{out}, sensor p increases h and broadcasts a new `HoleInfo` message. The timeout t_{out} is calculated as the time necessary for a sensor located $(h+1)$ hops apart to reach $Hex(p)$, that is $t_{\text{out}} = (h+1) \cdot 2R_{\text{s}}/v$, where v is the sensor speed.

Figure 4 illustrates the pull action performed by sensor p as described above.

6.2 Behavior of trigger forwarder sensors

When a sensor p receives a `HoleInfo` message and has not any slave to push toward the coverage hole, it participates in the pull activity by forwarding this message when necessary. In particular, it discards `HoleInfo` messages related to holes whose presence was already triggered by a snapped sensor q, unless they contribute additional information. Indeed sensor p evaluates new messages regarding a coverage hole previously advertised by sensor q only if they come from: 1) snapped sensors with *ord* value lower than $ord(q)$, or 2) snapped sensors with the same *ord* as q and hop counter h which is higher than the forwarding horizon issued by sensor q.

Case 1) happens when a new snapped sensor r detects the same coverage hole advertised by q, but the distance between p and r is lower than the distance between p and q. Case 2) happens when sensor q issues a new hole trigger demanding a forwarding horizon extension.

When processing `HoleInfo` messages, sensor p sets its order value equal to the adjacent sender order value increased by 1 and forwards the trigger to its adjacent snapped sensors only if $h > 0$. Such forwarded trigger message contains the updated status information of p and a hop counter decreased by 1. If sensor p receives several `HoleInfo` messages concurrently, it inserts them in a pre-emptive priority queue, where each message is treated with a priority that is inversely proportional to the distance from the coverage hole. As soon as the timeout of the `HoleInfo` message expires, sensor p selects the next element in the priority queue. Once the queue is empty it sets back its *ord* to the original value. This is necessary to stop the pull action after the coverage of the detected hole.

6.3 Sensors pushing redundant slaves

When a sensor p receives a `HoleInfo` message and finds an available slave to push towards the coverage hole, it updates the local information regarding its neighborhood. Thanks to the sequence of order value alteration, p finds a valid Moving Condition with respect to the direction of the coverage hole and properly starts a push activity.

6.4 An example

Figure 5 shows a typical scenario of the pull activity. The snapped sensor p detects a coverage hole in an adjacent position. Since p has no slaves in its hexagon and the Moving Condition with respect to its neighbors is unsatisfied, it starts the pull activity by setting its *ord* value to zero and broadcasting a `HoleInfo` message with null hop counter. Since sensor q does not have any slave to push toward p, at the expiration of the timeout, sensor p broadcasts another `HoleInfo` message increasing the previous hop counter. Sensor q evaluates the hop counter of the `HoleInfo` message it received from p and sets its own *ord* value to 1. Sensor q then forwards the trigger by broadcasting a `HoleInfo` message with decreased hop counter. Once again the timeout set by p expires because even sensor z has no slave to push, thus the procedure is repeated until the trigger, represented by the `HoleInfo` message, reaches sensor r which has an available slave s to push. Figure 1 shows the interleaved execution of the pull activity with the other algorithm activities. In particular, Figures 1 (e) and (f) show that node 7 starts the pull activity, as it detects a coverage hole and does not have any slaves to snap. To this end, it temporarily advertises a change of its *ord* function, which assumes the value of 0. In agreement with the pull activity, some nodes move towards the hole, as shown in Figure 1 (g). Figure 1 (h) shows that the node that started the pull activity has received a slave, so it sets back its *ord* function, and snaps the newly available slave. Figure 1 (i) illustrates several push actions (involving nodes 6, 9, and 4). The snapped nodes 10 and 12, that do not have any slaves, start their pull activity setting their *ord* function to 0. The pull activity going on in the left grid portion attracts some slaves towards the hole, as in Figure 1 (j).

7. P&P: MERGE ACTIVITY

The fact that many sensors act as starters implies the generation of several tiling portions with different orientations. The aim of the PUSH & PULL algorithm is to cover the AoI with a unique regular tiling thus minimizing overlaps of the sensing disks and enabling a complete and uniform coverage. Hence, the algorithm provides a merge mechanism to be executed whenever a sensor p receives a neighbor discovery message (`IAS`) from a snapped sensor q belonging to another tiling portion.

In this case, sensor p chooses to join the oldest grid portion (it discriminates this situation by evaluating the timestamp of the starter action, attached to any `IAS` message).

Notice that the detection of the sole neighbor discovery

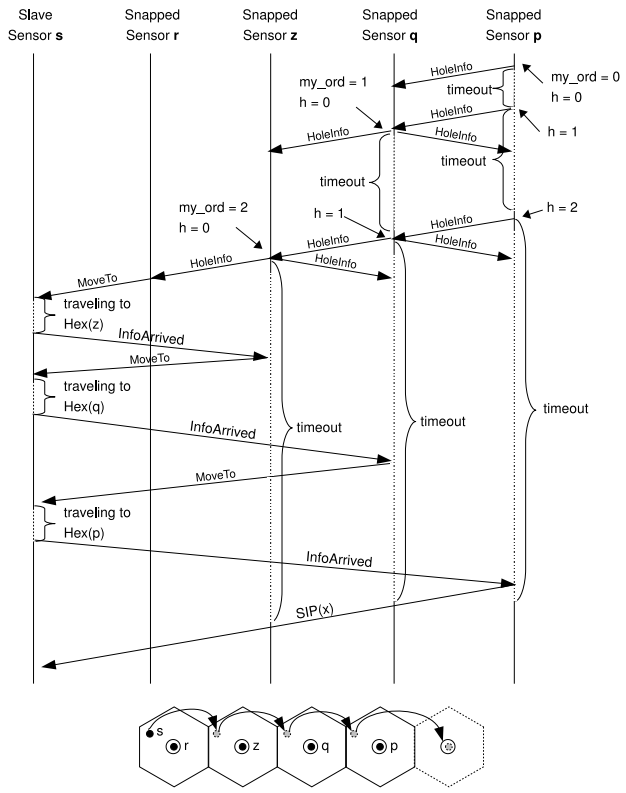


Figure 5: A typical scenario of the pull activity

messages is sufficient to ignite the tiling merge activity because such messages are sent after any tiling expansion and, if two tiling portions come in radio proximity to each other, at least one of them is increasing its extension.

In order to explain the grid merge activity, we refer again to figure 1. Figure 1 (j) shows the presence of two grid portions in radio proximity with each other. As a consequence of this reciprocal detection, the two grid portions start the tiling merge activity as shown in Figure 1 (k). In Figure 1 (l) the tiling merge activity is concluded and a unique grid is built.

In the following we give the details on the protocol implementation of the grid merge activity. We call $G_{\texttt{old}}$ and $G_{\texttt{new}}$ the tiling portions with lower and higher timestamp, respectively. We distinguish three possible cases.
1) Sensor p belongs to $G_{\texttt{new}}$ and receives a `IAS` message from q belonging to $G_{\texttt{old}}$. If sensor p is a slave, it switches its state to free or to slave of the sensor q depending on their mutual distance. Sensor p proactively communicates its new state to its neighborhood by sending either an `InfoFree` or an `InfoSlave` message. From now on p honors only messages from $G_{\texttt{old}}$ and ignores those from $G_{\texttt{new}}$.

This proactive communication of the new state of p is needed to advertise the presence of $G_{\texttt{new}}$ when there is no message activity within $G_{\texttt{new}}$ that is perceivable by the sensors in $G_{\texttt{old}}$. This way, the snapped sensor which p belonged to can properly update its slave set.

If p is instead a snapped sensor, it can not immediately switch to its new state because of its leading role inside $G_{\texttt{new}}$ (e.g. it leads the slave sensors in $S(p)$ and performs

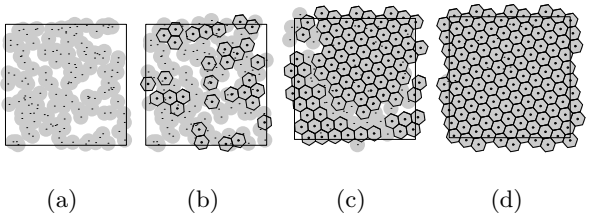


Figure 6: Deployment with random distribution

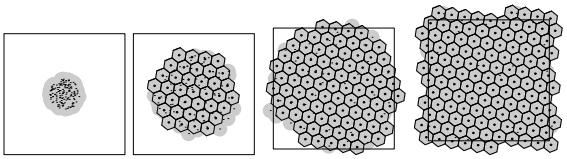
push and pull activities). Hence p temporarily assumes a hybrid role: it advertises itself as free/slave to the nodes of $G_{\texttt{old}}$ with an `InfoFree`/`InfoSlave` message and, at the same time, keeps on behaving as snapped node in $G_{\texttt{new}}$ until it receives a movement command (`SIP` or `MoveTo` message) coming from $G_{\texttt{old}}$.

If p received a `SIP` or a `MoveTo` command, p moves to the new snap position electing one of its slave in $G_{\texttt{new}}$ as a substitute with a `MoveToSubst` message. The selected slave should reply with a `SubstArrival` upon arrival to the snap position, within a given timeout. If this timeout expires before the reception of such `SubstArrival` message, p selects a new slave to snap. The process goes on until no more slaves are available. In this case p ceases its snapped role inside $G_{\texttt{new}}$ advertising its departure to its neighbors in $G_{\texttt{new}}$, broadcasting a `Retirement` message. Upon reception of a `Retirement` message the snapped neighbors that were located in positions adjacent to the one that p just freed, keep into account the new vacant position starting new snap activities. If otherwise, p receives a `SubstArrival` on time, it ceases its snapped role in $G_{\texttt{new}}$ and honors the commands issued by the snapped node in $G_{\texttt{old}}$.
2) Sensor p belongs to $G_{\texttt{old}}$ and receives a `IAS` message from q belonging to $G_{\texttt{new}}$: if p is a slave it ignores all messages from $G_{\texttt{new}}$. If p is snapped, it performs a neighbor discovery sending a `IAS` message, ignores all messages coming from $G_{\texttt{new}}$, apart from the neighbor discovery replies, and honors only messages from $G_{\texttt{old}}$. Observe that the neighbor discovery is necessary to ignite the merge mechanism and allows each snapped sensor in $G_{\texttt{old}}$ to collect complete information on nearby sensors that previously belonged to $G_{\texttt{new}}$.
3) Sensor p is free: sensor p honors only messages from $G_{\texttt{old}}$ and ignores those from $G_{\texttt{new}}$.

8. SIMULATION RESULTS

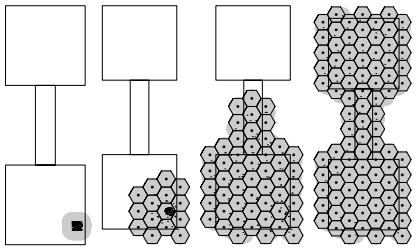
In this section we evaluate the performance of the P&P protocol through simulations. The parameter setting used in the experimental activity is as follows: $R_{\texttt{tx}} = 11$ m, $R_{\texttt{s}} = 5$ m, the sensor speed is 1 m/sec, squared AoI with size 80 m $\times$ 80 m.

In Figure 6(a) we show an example of the protocol execution starting from a random initial distribution of sensors over the AoI, whereas in Figure 7(a) sensors start from an initial deployment consisting of an high density region at the center of the AoI. In the two examples, the deployment obtained by P&P evolves through the configurations shown in (b) and (c), reaching the final deployment given in (d). Figure 8 gives a synthetic representation of how the sensor deployment evolves under P&P when 150 sensors are sent from a high density region in an AoI with a complex shape. In order to evaluate the performance of our protocol, we run some experiments with starting configuration depicted



(a) (b) (c) (d)

Figure 7: Deployment with central distribution



(a) (b) (c) (d)

Figure 8: Coverage of an irregular AoI

in figure 7(a), by varying the number of sensors. Results are average values calculated over 30 simulation runs.

Figure 9 represents the number of conflicting snap and push actions, averaged over the number of snap positions and of slave sensors involved in a push action, respectively. A snap conflict occurs whenever the same snap position is contended by two or more sensors being snapped, whereas a push conflict happens when a push offer made by one sensor becomes obsolete in consequence to push actions performed by other sensors.

The asynchronous behavior of P&P guarantees the resolution of the few snap/push conflicts that arise as a consequence of its distributed execution. Although growing with the number of available sensors, the average number of snap conflicts remains significantly smaller than 1, meaning that, in the considered scenarios, no more than one conflict happens per snap position. Similarly, when the number of sensors is larger than the minimum to guarantee the coverage completeness, the average number of push conflicts per slave sensor becomes almost stable at about 1.2 push conflicts per slave sensor. Figure 10 shows the coverage and termination time of the protocol execution. By coordinating distributed decisions and solving local conflicts, the P&P protocol guarantees the termination of the deployment in moderate time. Notice that after the coverage completion, P&P keeps on regulating some movements to uniform the redundant sensor density. The termination time evidences the capability of the P&P protocol to reach a final stable configuration, where neither movements nor message exchanges are performed.

The next figures detail the performance evaluation of our protocol in terms of energy consumption. The protocol activities having the major impact on the energy consumption are: movements, starting/braking actions and message exchanges.

Figure 11 shows the average moving distance per sensor. This contribution, although constant under growing number of sensors, is rather high as a consequence of the starting deployment that consists of a very dense sensor distribution at the center of the AoI.

An important contribution to the overall energy consumption is also due to the possible starting/braking actions performed by the moving sensors. Sensor movements indeed may be frequently interrupted to allow sensors make movement decisions such as changes in directions or role substitutions. Figure 12 shows that this contribution is also quite stable under an increasing number of sensors, evidencing a good scalability of the proposed approach.

The last term of the overall energy consumption is the number of message exchanges, shown in Figure 13. As the figure shows, this number remains stable even when the number of sensors increases significantly. It should be noted that both transmitting and receiving messages are energy consuming activities. Therefore, although Figure 13 evidences that the number of message exchanges is quite stable even under an increasing number of available sensors, the energy consumption related to this term can be also affected by the sensor density. Indeed, the higher the sensor density the higher the contribution to the overall energy consumption due to message receiving actions. This trend is made more evident in the following Figure 14, where we analyze the overall energy consumption. In this figure we utilize a unified energy consumption metric obtained as the sum of the contributions given by movements, starting/braking actions and communications. The energy spent by sensors for communications and movements is expressed in energy units. The reception of one message corresponds to one energy unit, a single transmission costs the same as 1.125 receptions [1], a 1 meter movement costs the same as 300 transmissions [12] and a starting/braking action costs the same as 1 meter movement [12].

Although generally moderate, the energy consumption has a minimum when about 250 sensors are available, showing that it is possible to optimize the energy consumption by looking for a trade-off in the number of sensors. On the one hand, having too few sensors implies that all the available sensors are strictly necessary to ensure the coverage completeness, and therefore each of them has to make several movements to reach its final position. On the other hand, having too many sensors makes each sensor receive plenty of messages from its neighborhood, causing an increase in the contribution to the energy consumption due to the message receptions. Nevertheless, it should be noted that in order to evidence the existence of such a minimum point, we had to significantly enlarge the scale of graph of the energy consumption. The trend of such parameter can be considered quite stable, especially if the operative setting is such that the ratio between the message reception and transmission costs is lower than the one considered in this paper, in which we study a critical case with respect to the parameter setting and the initial deployment.

9. CONCLUSIONS

In this paper we introduce P&P, a communication protocol that permits a correct and efficient coordination of sensor movements in agreement with the Push & Pull algorithm. Unlike previous works which introduce deployment algorithms without formalizing the related protocol, we address the realistic applicability of this approach. Indeed we deeply investigate the possible conflicts that may arise when

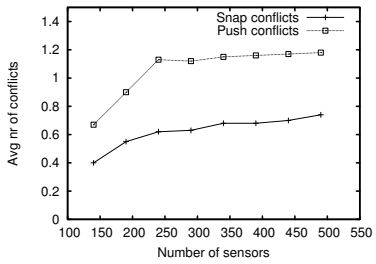


Figure 9: Snap and Push conflicts

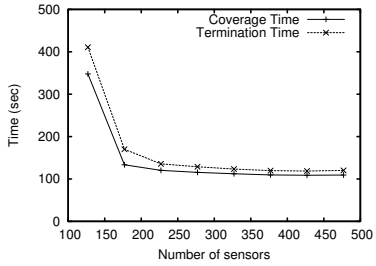


Figure 10: Termination and coverage time

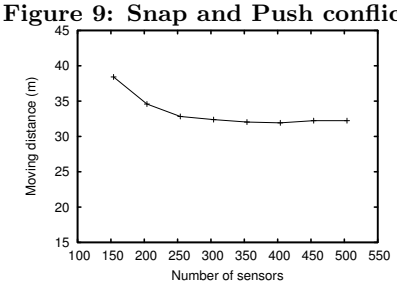


Figure 11: Traversed distance

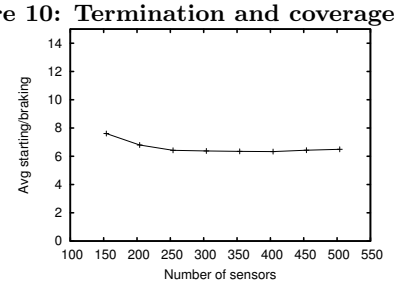


Figure 12: Starting/braking

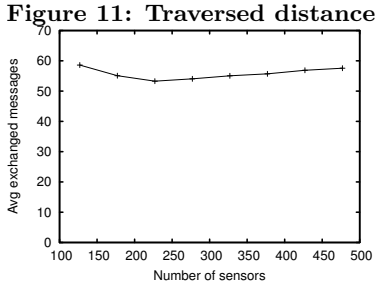


Figure 13: Exchanged messages

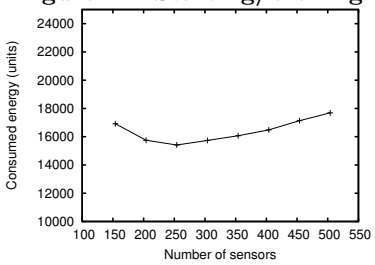


Figure 14: Consumed energy

asynchronous local decisions are to be coordinated, and propose protocol solutions.

Simulation results show the performance of our protocol under a range of operative settings, including conflict situations, irregularly shaped target areas, and node failures. These results evidence the protocol capabilities to fulfil the algorithm requirements and in particular termination, completeness and stability of the final coverage.

10. REFERENCES

[1] G. Anastasi, M. Conti, A. Falchi, E. Gregori, and A. Passarella. Performance measurements of mote sensor networks. *ACM MSWiM*, 2004.

[2] N. Bartolini, T. Calamoneri, E. Fusco, A. Massini, and S. Silvestri. Push & pull: autonomous deployment of mobile sensors for a complete coverage. *ACM Winet*, 2009.

[3] N. Bartolini, T. Calamoneri, E. G. Fusco, A. Massini, and S. Silvestri. Snap & spread: a self-deployment algorithm for mobile sensor networks. *IEEE/ACM DCOSS*, 2008.

[4] P. Brass. Bounds on coverage and target detection capabilities for models of networks of mobile sensors. *ACM Trans. Sensor Networks*, 2007.

[5] J. Chen, S. Li, and Y. Sun. Novel deployment schemes for mobile sensor networks. *Sensors*, 7, 2007.

[6] M. Garetto, M. Gribaudo, C.-F. Chiasserini, and E. Leonardi. A distributed sensor relocation scheme for environmental control. *ACM/IEEE MASS*, 2007.

[7] N. Heo and P. Varshney. Energy-efficient deployment of intelligent mobile sensor networks. *IEEE Trans. Systems, Man and Cybernetics*, 2005.

[8] W. Kerr, D. Spears, W. Spears, and D. Thayer. Two formal fluid models for multi-agent sweeping and obstacle avoidance. *AAMAS*, 2004.

[9] M. R. Pac, A. M. Erkmen, and I. Erkmen. Scalable self-deployment of mobile sensor networks; a fluid dynamics approach. *Proc. of IEEE/RSJ IROS*, 2006.

[10] S. Poduri and G. S. Sukhatme. Constrained coverage for mobile sensor networks. *IEEE ICRA*, 2004.

[11] G. Wang, G. Cao, and T. L. Porta. Proxy-based sensor deployment for mobile sensor networks. *IEEE MASS*, 2004.

[12] G. Wang, G. Cao, and T. L. Porta. Movement-assisted sensor deployment. *IEEE Trans. Mobile Computing*, 2006.

[13] Y. Zou and K. Chakrabarty. Sensor deployment and target localization based on virtual forces. *IEEE INFOCOM*, 2003.