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]


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.

N. Bartolini, A. Massini, S. Silvestri
Department of Computer Science
“Sapienza” University of Rome, Italy
E-mail: {bartolini, massini, simone.silvestri}@di.uniroma1.it
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 [15, 7, 5, 11] 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 [10] and [8]. Another methodology is based on the construction of Voronoi diagrams [14, 13]. 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 interactions. 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 \( h \) less or equal to the sensing radius \( R_s \). This setting guarantees both coverage and connectivity when the transmission radius \( R_{tx} \) is such that \( R_{tx} \geq \sqrt{3}R_s \).

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 \( VP(p) \) 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.
<table>
<thead>
<tr>
<th>Message name</th>
<th>Message fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>ID, coordinates, starter timestamp</td>
</tr>
<tr>
<td>InfoSnapped</td>
<td>ID, coordinates, cardinality</td>
</tr>
<tr>
<td>InfoSlave</td>
<td>ID, coordinates, energy level</td>
</tr>
<tr>
<td>InfoFree</td>
<td>ID, coordinates</td>
</tr>
<tr>
<td>SIP</td>
<td>ID, receiver ID, target position coordinates</td>
</tr>
<tr>
<td>AckSIP</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>ClaimPosition</td>
<td>ID, coordinates, timestamp</td>
</tr>
<tr>
<td>PositionTaken</td>
<td>ID, coordinates</td>
</tr>
<tr>
<td>InfoStopped</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>IAYS</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>CardinalityInfo</td>
<td>ID, cardinality</td>
</tr>
<tr>
<td>Offer</td>
<td>ID, receiver ID, cardinality, transaction ID</td>
</tr>
<tr>
<td>AckOffer</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>PositionTaken</td>
<td>ID, coordinates</td>
</tr>
<tr>
<td>AckInfoArrived</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>MoveTo</td>
<td>ID, receiver ID, dest. coord., dest. snapped sensor ID, trans. ID</td>
</tr>
<tr>
<td>InfoArrived</td>
<td>ID, receiver ID, transaction ID, energy level</td>
</tr>
<tr>
<td>HoleInfo</td>
<td>ID, hop counter, order value, hole coordinates, timeout</td>
</tr>
<tr>
<td>Subst</td>
<td>ID, receiver ID, energy level</td>
</tr>
<tr>
<td>AckSubst</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>SubstArrival</td>
<td>ID, receiver ID</td>
</tr>
<tr>
<td>ProfilePacket</td>
<td>ID, receiver ID, order value, priority queue, neighborhood info.</td>
</tr>
<tr>
<td>MoveToSubst</td>
<td>ID, receiver ID, order value, priority queue, neighborhood info.</td>
</tr>
<tr>
<td>Retirement</td>
<td>ID, hole coordinates</td>
</tr>
</tbody>
</table>

Table 1 Summary of the P&P messages

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 \( t_{\text{start}}(p) \) randomly selected over a time interval of length \( R_{\text{tx}}/v \), where \( v \) is the sensor movement speed. If at the instant \( t_{\text{start}}(p) \), 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 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 \textbf{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 \( L(p) \).

\subsection*{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 \textbf{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 \textbf{SIP}, and is available to fill the vacant position, it replies with an \textbf{AckSIP} message. This message contains the ID of the sensor that received the \textbf{SIP}, necessary for \( p \) to discriminate among the several sensors to which it sent \textbf{SIP} messages. If a sensor receives a \textbf{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 \textbf{SIP} message of \( p \) and lets the \textbf{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 \textbf{SIP} messages and the reception of the related \textbf{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 \textbf{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 \textbf{SIP} message for the position \( x \). Five cases may occur, described as follows.

1) Sensor \( p \) receives both the \textbf{AckSIP} and the \textbf{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 \textbf{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 \textbf{AckSIP} from \( q \) and a \textbf{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 \textbf{AckSIP} from \( q \) acknowledging its availability to fill position \( x \), but a failure occurred and the \textbf{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 \textbf{AckSIP} from \( q \), but receives a \textbf{IAS} message for position \( x \) from another sensor \( r \), before the expiration of the \textbf{AckSIP} timeout. Sensor \( p \) can update the local information regarding the snapped neighbor \( r \).

5) Sensor \( p \) does not receive the \textbf{AckSIP} from \( q \) nor the \textbf{IAS} from any other sensor within the \textbf{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 $\text{CardinalityInfo}$ message to its neighborhood. This message contains the ID and the virtual cardinality of $p$.

4.3 Behavior of the sensors being snapped

4.3.1 Sensor localization

A free sensor $q$ which receives a $\text{IAS}$ message, coming from a snapped sensor $p$, replies with either an $\text{InfoFree}$ or an $\text{InfoSlave}$ 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 $\text{InfoFree}$ message. If instead $q$ is located inside the hexagon of $p$, it switches its state to slave and replies to $p$ with an $\text{InfoSlave}$ 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 $\text{IAS}$ 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 $\text{SIP}$ message coming from a snapped sensor. Slaves reply only to $\text{SIP}$ messages coming from their related snapped sensor, while free sensors only reply the first $\text{SIP}$ message they receive and ignore subsequent ones.

After sending the $\text{AckSIP}$ 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 $d \leq \sqrt{3}h/2$.

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 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 PositionTaken 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 \texttt{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 \texttt{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 \textit{AckSubst} message.

If sensor \( r \) receives an \textit{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 \textit{SubstArrival} message containing the same
fields of the \textit{AckSubst} message. Sensor \( p \) replies to \( r \) with a \textit{ProfilePacket} message
that is necessary to enable a complete role exchange and starts travelling towards the
destination.

If sensor \( r \) does not receive an \textit{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 \textit{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
\textit{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 \textit{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 \textit{Offer} message
to \( q \). Notice that the \textit{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 \textit{AckOffer} message. Sensor \( q \) receives the \textit{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 \textit{Offer} message. As the Moving Condition is still
satisfied, \( q \) replies with an \textit{AckOffer} message, incrementing its virtual cardinality and
broadcasting a \textit{CardinalityInfo} message.

When node \( q \) receives the \textit{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 \textit{AckOffer} message from \( q \), thus it selects \( r \) within its slaves,
and send it a \textit{MoveTo} message. Sensor \( r \) moves towards the hexagon of \( q \), and sends an
\textit{InfoArrived} message as soon as it arrives. Sensor \( p \) sends a \textit{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 un-snapped 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

InvitationAcceptance Invitation position-taken message per il caso in cui l’invitante avvisa tutti che ha gi trovato l’accordo con uno slave per tappare il buco. Selection PositionTake serve per notificare eventuali altri inviter che lo slave sta arrivando a coprire quel buco, infatti l’inviter non fa la snap perch lo slave va diretto al buco.

****

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 sends an invitation message that will attract slave sensors from nearby regions to fill the hole. For this reason, the sensors having this role are hereafter called Inviters.

The second role is performed by the slave sensors which receive the aforementioned invitation messages. These sensors, hereby called Invited, reply to the inviter to communicate their availability to move and fill the hole.

Such an invitation mechanism only works if available slave sensors can be found in the radio proximity of the inviter. By contrast, when there are no such sensors, the inviter proceeds inviting new slaves by means of a limited distance broadcast message at larger and larger distances.

Therefore a third role is performed by snapped sensors which receive the broadcast invitation messages. The sensors having this role act as forwarder of the invitation, when necessary, in order to reach hexagons with redundant slaves that can fill the holes. These sensors will be named Forwarders.

Notice that multiple invitations may be received by the same slaves. Such messages are inserted in a priority queue by the invited slaves, ordered on the basis of the distance between the hole and themselves.

If the inviter receives multiple replies from several available slave sensors, it selects the one with lower distance to the hole.

The protocol according to which fixed sensors invite the available slaves is partly inspired by the invitation mechanism introduced in [12]. Such a pull mechanism is specifically designed to avoid the case of several invited sensors moving to the same destination.

In the following we summarize the invitation mechanism.

1. A snapped sensor detecting a hole makes an invitation to advertise the presence of a vacant position in adjacent potential snap positions. This message traverses the network according to a limited distance broadcast, whose extent can be increased in subsequent time slots for successive reattempts, until the inviter receives an acknowledgement from at least one invited slave sensor.
2. A slave collects the invitations received in a given time interval. If it receives multiple invitations, it stores them in a priority based queue, where the priority is the distance to the destination. The lower the distance the higher the priority.

   An invited slave acknowledges only the highest priority invitation, so as to notify the inviter that it is available to fill the vacant position.

3. The inviter that receives such an acknowledge from one or more slaves, selects the one that is closer to the hole, and notifies the others that an agreement to cover the hole was already made with another invited slave.

4. The invited slave that receives an acknowledge from the inviter starts moving to the destination. On the contrary, the invited slaves that receive a position-taken notification, process the next element in their invitation queue.

6.1 Sensors detecting coverage holes

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

   If $p$ can not receive any sensor from its snapped neighbors, it starts the pull activity. To this purpose $p$ broadcasts an Invitation message containing its ID, a hop counter $h$, and the vacant position coordinates.

   The hop counter $h$ represents the forwarding horizon of the Invitation message. Initially $h$ is set to zero, thus the snapped sensors receiving an Invitation do not forward this message. After a given timeout, if $p$ does not receive any acknowledge, it evaluates the opportunity to enlarge the hop counter and send a new invitation.

   On the contrary, if the inviter $p$ receives an InvitationAcceptance from one or more available slaves, it selects the sensor $s^* \in R$ which is the closest to the vacant position among the sensors in $R$ that is the set of the sensors that replied in a time window of length $\tau$. The sensor $p$ sends a message SlaveSelected (multicast message at maximum $h$-hop) to all the sensors which accepted the invitation. It also sends a PositionTaken in order to avoid other snapped sensors trying to initiate another pull activity to fill the same hole.

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

6.2 Behavior of the invited sensors

   When a slave sensor $s$ receives an Invitation message, and has not already committed to fill a different hole, it inserts such invitation message in a priority queue, containing all the invitation received in a given time interval $\tau$. This time interval starts as soon as a slave sensor receives the first invitation. All the invitation messages received within this interval are inserted in the queue according to a priority based on the distance from the invited slave to the destination. At the end of this time interval, the invited sensors processes the elements in the priority queue one by one. If another slave sensor has already accepted the invitation to fill the hole being considered (this event corresponds to $s$ receiving a SlaveSelected message containing the same hole coordinates but another invited id), $s$ proceeds by considering the next element in its priority queue.
The sensor \( s \) sends an \textit{InvitationAcceptance} to the inviter related to the highest priority element in the queue, for which no previous agreement has been notified. As soon as \( s \) receives a \textit{SlaveSelected} message regarding the same hole, it analyzes the field containing the id of the selected invited slave. If \( s \) is the selected slave, it starts its movement towards the destination and fills the hole. Otherwise, \( s \) drops the record containing the information on the just considered hole, and goes on by processing the next element in the queue.

Notice that, whenever \( s \) receives a \textit{SlaveSelected} message (with invited id different from \( s \)) related to a hole for which there is an element in its priority queue, it removes the above element from the queue to take into account that an agreement to fill that hole was already made with another invited slave sensor.

### 6.3 Behavior of forwarders snapped sensors

When a snapped sensor \( p \) receives an \textit{Invitation} message it participates in the pull activity by forwarding this message when necessary. In particular, it discards \textit{Invitation} messages related to holes whose presence was already notified by another snapped sensor \( q \).

The sensor \( p \) forwards the invitation to its adjacent snapped sensors only if \( h > 0 \). The hop counter of the forwarded invitation will be decreased by 1.

### 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 \textit{ord} value to zero and broadcasting a \textit{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 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 $G_{old}$ and $G_{new}$ the tiling portions with lower and higher timestamp, respectively. We distinguish three possible cases.

1) Sensor $p$ belongs to $G_{new}$ and receives a IAS message from $q$ belonging to $G_{old}$. 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 $G_{old}$ and ignores those from $G_{new}$.

This proactive communication of the new state of $p$ is needed to advertise the presence of $G_{new}$ when there is no message activity within $G_{new}$ that is perceivable by the sensors in $G_{old}$. 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 $G_{new}$ (e.g. it leads the slave sensors in $S(p)$ and performs push and pull activities). Hence $p$ temporarily assumes a hybrid role: it advertises itself as free/slave to the nodes of $G_{old}$ with an InfoFree/InfoSlave message and, at the same time, keeps on behaving as snapped node in $G_{new}$ until it receives a movement command (SIP or MoveTo message) coming from $G_{old}$.

If $p$ received a SIP or a MoveTo command, $p$ moves to the new snap position electing one of its slave in $G_{new}$ 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 $G_{new}$ advertising its departure to its neighbors in $G_{new}$, 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 $G_{new}$ and honors the commands issued by the snapped node in $G_{old}$.

2) Sensor $p$ belongs to $G_{old}$ and receives a IAS message from $q$ belonging to $G_{new}$: if $p$ is a slave it ignores all messages from $G_{new}$. If $p$ is snapped, it performs a neighbor discovery sending a IAS message, ignores all messages coming from $G_{new}$, apart from the neighbor discovery replies, and honors only messages from $G_{old}$. Observe that the neighbor discovery is necessary to ignite the merge mechanism and allows each snapped sensor in $G_{old}$ to collect complete information on nearby sensors that previously belonged to $G_{new}$.
8 Experimental results

In this section we compare through simulations the performance of the P&P protocol to a previously proposed virtual force based algorithm. To this aim, we developed a simulator on the basis of the wireless module of the Opnet software [16]. Before introducing the simulation results, in the following Section we briefly describe the virtual force based algorithm used for comparisons.

8.1 Parallel and Distributed Network Dynamics (PDND) algorithm

The Parallel and Distributed Network Dynamics (PDND) algorithm [9] is a virtual force based approach according to which the force exerted by the sensor $s_i$ on the sensor $s_j$ is modelled as a piecewise linear function. It is repulsive when the distance between $s_i$ and $s_j$ is lower than an arbitrarily tuned parameter $r^*$; it is attractive when the distance is larger, until it vanishes at another arbitrarily set distance. In order to ensure the convergence of PDND, the formulation of this force must respect the condition of Lipschitz continuity. In this case, the single sensor movement is limited by an upper bound that guarantees that the potential energy is always decreasing, hence avoiding oscillations. The PDND algorithm is a round based algorithm. At each round the sensors initially exchange their position information, calculate the resulting force and then move accordingly. The algorithm is proved to achieve a final stable configuration in which all sensors stop moving provided that a positive minimum moving threshold is set.

8.2 Simulation results

In this Section we describe the performance comparisons between P&P and PDND. The parameter setting used in the experimental activity is as follows: $R_{tx} = 11$ m, $R_s = 5$ m, the sensor speed is 1 m/sec, the AoI is a square with size 80 m x 80 m. For the PDND algorithm, the round length is set to 1 sec, the threshold $r^*$ to $2R_s$ while the minimum moving distance is set to 0.1 m, as in [9].
Before giving a quantitative evaluation of the protocol performance, we show some examples of the execution of P&P and PDND. In Figure 6(a - d) we show an example of the P&P protocol execution starting from a random initial distribution of 150 sensors over the AoI, whereas in Figure 7(a - d) we show the execution of PDND starting from the same initial configuration. Similarly, in Figures 8(a - d) and 9(a - d) we show the execution of P&P and PDND, respectively, starting from a distribution where 150 sensors are densely deployed at the center of the AoI.

The protocol P&P is able to achieve a complete coverage of the AoI in both scenarios. It is to notice that, in the case of randomly deployed sensors, several tiling portions are created at the beginning of the algorithm execution (Figure 6(b)), due to the random election of the tiling starters. As the tiles keep growing, and arrive in radio proximity to each other, the merge activity starts, giving precedence to older tiles, which are likely to be composed by a higher number of nodes (Figure 6(c)). At the end of the algorithm execution, only one tile has remained which entirely covers
the AoI (Figure 6(d)). On the contrary, when the initial deployment is dense as the one shown in Figure 8(a), all nodes are able to communicate thus only one tile is created 8(b - c).

The PDND algorithm is also able to cover the AoI in the considered scenarios. Nevertheless, as we will show in the following, it requires much longer time to achieve the final deployment and consumes a higher amount of energy with respect to P&P.

In order to conclude this qualitative evaluation, we show in Figure 10 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. Also in this case P&P is able to achieve a complete coverage of the AoI.

In order to compare the performance of our protocol to the ones of PDND, we run two set of experiments starting from two different initial sensor deployments, by varying the number of deployed sensors. In the following experiments we increase the number of deployed sensors from 150 to 550. We do not show the coverage achieved by the two algorithms as in the considered interval of available sensors, both algorithm always reach a complete coverage.

In the first set of experiments we considered the random initial deployment depicted in Figure 8(a). Figure 17 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 18 shows the termination time of P&P and PDND, i.e. the time at which all sensors stop moving. According to the protocol P&P this situation occurs when all the activities are terminated. In particular, the snap and pull activity terminate
as soon as all the snapping position are occupied. The push activity terminates when
the moving condition is not satisfied among all the adjacent snap sensors. Finally,
the merge activity ends as soon as there is only one tile in the network. We have
formally proved in [2] that such a final stable configuration is always reached by the
PUSH&Pull algorithm, here we experimentally show that the P&P implementation
of the PUSH&Pull algorithm satisfies the algorithm requirements, guaranteeing the
achievement of a stable state after a finite time. The PDND algorithm is also proved
to converge to a stable configuration in which the forces acting to all the sensors are
balanced. Nevertheless, as Figure 18 points out, PDND requires a time one order of
magnitude longer with respect of the one needed by P&P. By coordinating distributed
decisions and solving local conflicts, the P&P protocol guarantees the termination of
the deployment in moderate time. Both algorithms require a longer time to terminate
when 150 sensors are available. This is due to the fact that this number is closer to
the minimum number of sensors required to entirely cover the AoI. Nevertheless, the results highlight a good scalability of the P&P with respect to the number of sensors, as the time needed to terminate slightly decreases by increasing the number of available sensors.

The next figures detail the performance evaluation of the two protocols 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 19 shows the average moving distance per sensor. Both algorithms show a decreasing behaviour of the moving distance by increasing the number of available sensors. This is due to the initial random distribution which ensures an even density over the AoI that helps the two algorithms in converging to the final configuration. Nevertheless, P&P let sensors traverse shorter distances with respect to PDND.

An important contribution to the overall energy consumption is also due to the starting/braking actions performed by the moving sensors [14]. Figure 20 shows that the PDND algorithm requires orders of magnitude more starting/braking actions with respect to P&P. Indeed, according to PDND, in order to guarantee that the potential energy of the system decreases with time, sensors are allowed to move only for a very short distance at each round, resulting in a very high number of moving actions. On the contrary, P&P makes precise movements, drastically reducing the number of starting/braking actions. Moreover, as Figure 20 shows, the number of moving actions decreases 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 21. The round based nature of PDND, according to which sensors have to exchange their position information at each round, results in very high number of exchanged messages as the algorithm requires a very long time to terminate. On the contrary under P&P, as the Figure shows, the number of exchanged messages remains almost 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 21 evidences that the number of message exchanges is quite stable under both algorithms, 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 22, 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 1 meter movement costs the same as 300 transmissions [14] and a starting/braking action costs the same as 1 meter movement [14].

As expected by the above comparisons in the energy consuming activities between P&P and PDND, the former shows an overall energy consumption which is two orders of magnitude less than the one of PDND. It is to notice that the overall energy consumption of P&P is even lower than the one required by PDND only for communications.

In the second set of experiments we compare the performances of the two algorithms starting from the initial deployment shown in Figure 8. It is to notice that,
with respect to the previous experimental setting, the P&P algorithm shows an higher number of conflicting actions. Indeed, by starting from a denser deployment, more sensors contend for the same snapping positions and similarly more sensors are pushed towards the same hexagons. The termination time increases and so does the number of exchanged messages. This is due to the fact that, in order to achieve a uniform deployment, more sensors have to be moved, and for longer distances, from their initial positions, with respect to the random deployment case. Nevertheless, similarly to the previous experimental set, P&P outperforms PDND. In particular, the PDND algorithm terminates in a much longer time with respect to P&P and shows an overall energy consumption which is two orders of magnitude higher than P&P.
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 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 fulfill the algorithm requirements and in particular termination, completeness and stability of the final coverage.

References