Robust sensor self-initialization

Whispering to avoid intruders

Carlos Ribeiro

IST / INESC-ID

Abstract. Wireless Sensor Networks (WSN) are becoming bigger and with this growth comes the need for new automatic mechanisms for initializations done by hand. One of those mechanisms is the assignment of addresses to nodes. Several solution were already proposed for mobile ad-hoc networks but they either: i) do not scale well for WSN; ii) have no energy constraints; iii) have no security considerations; iv) or have no mechanisms to handle fusion of network partitions.
We proposed an address self-assignment protocol which: uses negative acknowledgements and an improved version of a flood control mechanism to minimize the energy spent; uses a technique named *whispering* to achieve robustness against malicious nodes; and uses *private nicknames* to handle fusion of network partitions.

1 Introduction

Wireless Sensor Networks (WSN) have been arousing the interest of both researchers and the general community. WSNs are networks composed of small and cheap devices with sensing abilities which are able to communicate with each other through radio signals. The combination of sensing and radio communication abilities makes these networks ideal to build distributed sensing networks where each node collaborates by sensing one or more phenomena in its neighborhood and relaying it to a central node.

In order to be cheap and last for long periods without management sensor nodes have several challenging constraints, from which the most important one is energy. Thus every algorithm and protocol designed for sensor networks should always be energy conservative.

Given that sensor networks should be deployed on every kind of environment, including very hostile environments, security should be a major concern. Usually, achieving security implies some energy loss. However, this loss should be kept to a minimum when there is no threat to defend against.

One of the problems of sensor networks is the naming. Given that a sensor network could be comprised of a large amount of nodes, the unique addressing of each node may be a problem. Currently, nodes are initialized by hand with a unique number when the code is uploaded to the sensor node. This solution is simple and energy efficient but, it requires the tedious task of programming and initializing every node. In the initial versions of wireless sensors' operating systems, every sensor had to be programmed individually through physical contact using a special programming device. In those days initialization was not a big issue because it could be easily done with the programming. However, currently, wireless sensors are being programmed using their wireless network [1,2], which makes naming much more difficult.

Wireless programming has two advantages over physical contact programming. It scales better because there is no need to physically move the sensors to the programming device and sensors can be reprogrammed after deployment to correct programming mistakes, to adapt the sensor to a new environment or, simply, to upgrade the software with an improved version.

Given that sensor programming is currently done by wireless radio, and that wireless radio communications require addressing each individual sensor, then naming can not be piggybacked on sensor programming as it used to be.

1.1 Zero message solutions

An obvious solution would be for each sensor to choose a random ID for itself. This solution is fast, energy efficient and safe. It is fast and energy efficient because it does not involve any messages, and it is safe because there is no way, for an attacker, to prevent a sensor from choosing an ID.

However, for this solution to work we would need an ID space of at least 32 bit (preferably 64 bit) and, currently, most wireless sensor communication protocols use 16 bit IDs, mostly because 64 bit IDs would make communication headers to big for such devices. In TinyOS, the usual payload length is 29 bytes, and the maximum packet size in 802.15.4 radio is 128 bytes [3], thus it would be impossible to use 16 bytes just for addressing purposes (8 bytes for the receiver and 8 for the sender).

With 16 bit IDs the random solution does not work. In such a situation the collision probability (i.e. the probability of two nodes choosing the same ID) is given by the birthday paradox $p(X(n_t)) = 1 - \prod_{i=1}^{n_t-1}(1 - \frac{i}{2^{16}})$, in which X is a discrete uniform distribution of IDs and n_t is the total number of nodes deployed. As it can be seen in Figure 1, the collision probability is over 10% with only 120 nodes, and reaches 50% with $\sim$ 300 nodes.

It can be argued that node's IDs need to be unique only in their vicinities, i.e. only the nodes with common direct neighbors need to have unique IDs [4]. Surely this is true but it does not help much. In such a situation the collision probability can be given by $p(X(n_t, \overline{n_v})) = 1 - (1 - \frac{\overline{n_v}}{2^{16}})^{n_t - \overline{n_v} - 1} \prod_{i=1}^{\overline{n_v}-1}(1 - \frac{i}{2^{16}})$, where n_t is the number of nodes deployed and $\overline{n_v}$ is the average number of nodes directly reachable by each node. As it can be seen in Figure 2 the number of nodes that need to be deployed to have a 10% collision probability with an average of 20 neighbors is just 356.

Another solution would be to use a 64 bit unique manufacturer number to derive the 16 bit ID of each node. But that would require a collision free transformation function from every 64 bit ID to a 16 bit one, which is not possible for the same reasons that the random solution does not work.

Thus we need to develop a protocol to ensured collision free IDs. Global collision free IDs are difficult to ensure, and most of the time they are not

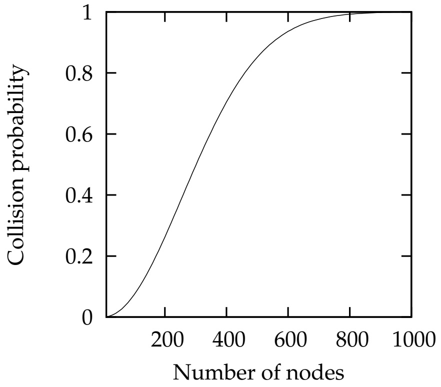

Fig. 1. Collision probability with the number of nodes deployed.

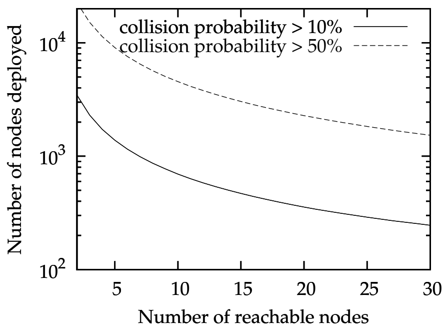

Fig. 2. Number of nodes deployed for achieving a 10% and 50% collision probability with the average number of nodes directly reachable by each node.

needed [4,5]. In WSNs, nodes are usually addressed by data attributes and not by their unique global addresses. For instance, in the direct diffusion protocol [6], communication is data centric. A node requests data by sending interest to named data. This interest is propagated to its neighbors building an interest tree. Whenever a source needs to send data, the data flows hop by hop over that interest tree to all nodes that have manifest interest in it.

1.2 Proposal

We have developed a message efficient and secure protocol to ensure the distribution of local unique IDs among neighbor nodes. The protocol is more efficient in terms of number of messages than similar ones, it is secure against a bounded number of malicious nodes and is able to handle late deployment scenarios and partition rejoining without resetting established addresses.

The protocol efficiency is obtained through the use of only negative acknowledgements and through an improvement over a previously proposed flooding control technique. The security features of the protocol are obtained without cryptography, through a technique named *whispering*. We are assuming that sensor nodes are sold without key material in place. If cryptographic keys are going to be needed, they will be distributed later over the wireless medium together with sensor programming. Finally, the solution to handle partition rejoining is accomplished using *private nicknames* between sensors.

We are going to describe some related work in Section 2 and the basic protocol solution in Section 3. In Section 4 we describe the solution to avoid intruders and in Section 5, we handle the incremental deployment of new sensors. Finally, in Section 6 we present our conclusions and describe future work.

2 Related work

The naming problem which we intend to solve has been addressed before for WSNs and for Mobile Ad-hoc Networks (MANET). The IETF Zeroconf working group proposed a solution for MANETs [7] which rely on the discovering abilities of the underlying routing protocol. In this proposal each node independently chooses an address and then sends a routing requesting packet to that address. If a route is found within a timeout period, the address is already in use; otherwise, the address is not used and the protocol ends. The main problem of this protocol is the definition of the timeout when the number of hops needed to reach every node in the network increases. The protocol was developed for MANETs and do not scale well for WSNs where the number of nodes and hops between them is much higher.

Unlike the Zeroconf working group proposal, most naming solutions' goal is to find a unique 2-hop ID. This problem is known as the neighborhood unique naming (NUN) problem, and is similar to the classical coloring graph problem with conditions at distance 2. In [8] it is proven that *there is no determinist self-stabilizing algorithm to solve the NUN problem in uniform and anonymous networks under distributed scheduler* and proposes a self-stabilizing probabilistic algorithm. The algorithm is very simple. Each node keeps two variables, one with its ID and one with the ID of two colliding nodes in its neighborhood. If there are no collisions in the neighborhood, the second variable is empty. Each node starts by asking every neighbor their ID to calculate the second variable. It then asks its neighbors for their variables values. If any of these values is equal to its own ID the node randomly chooses another. The algorithm was proven to self-stabilize, although no protocol was given to implement it. In particular, it is not clear how messages from two distinct nodes with the same ID can not be confused with a repetition of a previous message.

The same strategy is followed in [9], but instead of using the 2-hop neighborhood it uses a 3-hop neighborhood and a cache in every node to keep the IDs of its 3-hop neighborhood. It claims, that by using the 3-hop neighborhood, it bounds each node number of attempts to choose an ID. However, no consistency protocol for the 3-hop neighborhood cache is given, which makes it difficult to calculate the average number of messages required to reach a consistent state.

A cache is also used in [4] to keep IDs of direct neighbors. In this proposal, each node sends a periodical message with its ID. This message is stored in the cache of its neighbors. If a node detects that two of its neighbors have the same address, it sends a warning message to one of them. With this protocol, nodes may change addresses several times during the life-time of the network which may not be acceptable by every application or routing protocol. Moreover, the periodic broadcasting of IDs may be too energy expensive, and the authors fail to prove the self-stabilization of the protocol.

The approach followed in [10] is different but also probabilistic. They leverage on the wireless nodes' ability to detect media access collisions to know if there are other nodes contending for an ID or not. If a node discovers that no one else is broadcasting at the same time, it takes the ID for itself and every one else knows that that ID is taken. If several nodes broadcast at the same time, they all flip a coin to decide if they will participate in the next round. On average, only half of the contenders transmit in the next round. After several rounds only one node will transmit, and will get the ID. Although simple, this solution assumes that the radio is able to listen at the same time it transmits, which is not true in most inexpensive radio transmitters.

With the exception of the solution described in [11] most 2-distance graph coloring algorithms and address assignment protocols are either deterministic and semi-centralized or distributed and probabilistic. The reason why distributed protocols are probabilistic is the fact that, under a distributed and unfair scheduler, every node may precisely copy all the other nodes' movements always choosing the same IDs, thus the algorithm may never end. Clearly a deterministic solution is better than a probabilistic one, because there is always the possibility that it never ends. However, most deterministic solutions do not scale well because they are either centralized or semi-centralized.

The centralized solution is never used in MANETs or WSNs. It would be similar to having a DHCP server replying to every node, which clearly does not scale beyond a few dozens of nodes. The semi-centralized solutions works by starting the assignment process at one specific central node and then distributing the assignment workload among other nodes. The DRCP and DAAP protocols [12] work together to assign addresses in MANETs. The DRCP is used by the node requesting an address as in DHCP: the node starts by asking if any of its neighbors is acting as a DRCP server and if some of them reply, it chooses one of them to get the address from. After having received the address, the node uses the DAAP protocol to ask its DRCP server for half of its pool of addresses, and then proceed by acting as a DRCP server.

The ZigBee$^{\mathrm{TM}}$communication protocol uses two types of addresses: 64 bit global unique addresses and 16 bit network unique short addresses. The 64 bit addresses are used at the beginning of the network deployment to establish the 16-bit addresses, which are used thereafter. The protocol which establishes the 16-bit addresses is similar to DRCP/DAAP. However instead of using two distinct steps for assigning an address and for assigning a pool of addresses, ZigBee$^{\mathrm{TM}}$only uses one; and instead of giving up half of its address space to each child node, a node equally divides its pool of addresses among its neighbors. Neither DRCP/DAAP or ZigBee$^{\mathrm{TM}}$address assignment protocols scale well when the number of nodes is too large or the address space is too small.

The solution presented in [11] does not have this problem but it is costly in terms of time. In essence, the solution uses a token to establish an order between node's color changing, which in a network of several hundred nodes may take some time. Moreover, this solution requires the synchronized update of state variables in both sender and receiver nodes. This is a problem when nodes do not have a valid address, because in such situations it is difficult to establish a single receiver.

Finally, most 2-distance graph coloring algorithms [11,13,14] try to find the graph coloring which uses the minimum number of colors. We have a much more relaxed goal. We want to find a 2-distance graph coloring with a minimum number of messages, bounded by a maximum of 2^{16} colors.

3 Basic protocol

The basic protocol objective is twofold; i) ensure a unique local identification on the WSN over a distance 2 neighborhood with an arbitrary large probability $p < 1$ and, ii) minimize the energy loss by minimizing the number of messages sent and received.

The protocol assumes no local or global knowledge of topological information. This includes global and relative geographic coordinates, number of neighbors, local and global density, or even the global number of nodes. This is important in a scenario where most sensor nodes do not have a GPS module, and are distributed randomly over the sensor field. In such scenario, it is not possible to know geographic information at every sensor without running a localization protocol, which can only be run if proper addressing is in place. Therefore, although topological information may be acquired in the future, it is not available at initialization time.

Unlike several initialization protocols [8,9], we have chosen to keep state variables private to each node, i.e. we avoid the use of caches with partial knowledge of the state variables of other nodes. Although such caches would improve the nodes' knowledge over their neighborhood, we avoid expensive cache coherence protocols.

The basic protocol is very simple, each node chooses a random ID for itself and asks its neighbors if they have chosen the same ID. If at least one of them has chosen the same ID, it replies with a NACK, saying that a collision was found, otherwise each receiving node rebroadcasts the query to its own neighborhood. The nodes receiving these rebroadcasts check the receiving packet for a collision with their own IDs. If they find a collision, they reply in the same way as the first hop nodes do, otherwise they do nothing (A complete description of the protocol can be found in Appendix A).

Notice that there are no positive replies; only negative ones. This is because the probability of finding a collision in a 2-hop neighborhood is very low, thus in the usual scenario only query messages are sent. The collision probability in a 2-hop neighborhood is given by $p_c = 1 - (1 - 2^{-16})^{4\overline{n_v}}$, where $\overline{n_v}$ is the average number of neighbors[1], which is $\sim 10^{-3}$ for a 20 node neighborhood.

The first problem that the protocol needs to overcome is how to distinguish the rebroadcast messages originated in itself from the ones originated in other sensors. If a node trying to establish an ID receives a query for that same ID, it should answer declaring that that ID has been taken, even if that action results in neither of the nodes sticking with the ID. However, if the node is hearing an

[1] Note that, if $\overline{n_v}$ is the average number of nodes within a 1-hop neighborhood, then, assuming a uniform density distribution, the average number of nodes in a 2-hop neighborhood is $4\overline{n_v}$.

echo of its own query, it should do nothing. Thus we need a way to uniquely identify the messages.

The messages sent by each node are stamped with a collision free 64 bit node identifier (extended ID). This extended ID can be a manufacturer unique number, when available, or a random number generated whenever a node starts. However, as we will describe later a random number is preferred over a manufacturer unique number for security reasons. Note that extended IDs are only used in the context of the initialization protocol. Afterwards, only 16 bit IDs are used. In fact, the protocol can be seen as a recoloring protocol with a smaller color space.

Two other similar problems happen when a rebroadcast node needs to relay a NACK back to the original querying node, and when a node receives a NACK for its own ID. In both cases nodes should only act upon NACKs which were triggered by their own queries, otherwise the protocol may not stabilize.

Self-stabilization, as defined in [15], is an important property of a distributed protocol. It ensures that regardless of the initial state of the system and regardless of the scheduling of actions taken by each participating node, the system will reach a legitimate final state in a finite number of steps.

Beauquier *at al.* [16] redefined self-stabilization for probabilistic protocols in such a way that regardless of the initial state of the system and regardless of the scheduler strategy, the system will reach a legitimate final state with probability 1. Using the framework for proving self-stabilization of probabilistic protocols, defined in [16], it can be shown that the previously described protocol, satisfies the above mentioned definition of self-stabilization, if extended IDs are used to link queries and replies.

Informally, the framework, defined in [16], states that a probabilistic protocol is self-stabilizing for a given specification if there is a sequence of predicates over system states $L_i(S) \dots L_n(S)$ where S is a system state and $n > i \geq 0$ that:

- The last predicate $L_n(S)$(known as the legitimate predicate) of the sequence is a predicate that identifies a legitimate final state according to the specification.
- For every scheduler, the probability of reaching a system state satisfying the specification from a state verifying the legitimate predicate is 1, which can be formalized by the following conditional probability.

$$P(L_n(S_{m+k}) \diagup L_n(S_m)) = 1, k > 0, m \geq 0$$

- For every scheduler strategy, if the probability of reaching a state verifying one predicate in the sequence is 1, then the probability of reaching a state verifying the next predicate in the sequence is also 1, which can be formalized by the following conditional probability.

$$P(L_{i+1}(S_{m+k}) \diagup L_i(S_m)) = 1, k \geq 1$$

The first two are easily verified by the protocol. If we choose n to be the total number of nodes, and $L_i(N) \vdash \{N_{cf}(S) \geqslant i\}$, where $N_{cf}(S)$ is the number of nodes with a collision free ID in state S, the last predicate $(L_n(S))$ clearly identifies a legitimate final state (first requirement). Moreover, after reaching a legitimate state (i.e. every node has a 2-hop unique identifier) the protocol ceases to send NACKs. Since IDs are only changed when a NACK arrives, the system will reach a final configuration verifying the specification (second requirement).

To prove that the protocol satisfies the last requirement, we will use another result from [16]. It states that the third requisite is verified if both predicates are *closed* and verify the local convergence property. Two predicates are said to verify the local convergence property if, according to a scheduling strategy, the probability of reaching a state verifying the second predicate from a state verifying the first predicate, in less than $k > 1$ steps, is greater than $\delta > 0$.

The predicate $L_i(N) \vdash \{N_{cf}(S) \geqslant i\}$ is closed under the given protocol, because whenever a node chooses a 2-hop unique identifier and every 2-hop neighbor has the opportunity to reply and does not do it, the node never changes its ID again provided that it only acts upon NACKs to its own queries. If the node tries to optimize the process of detecting a collision by overhearing NACKs to queries initiated by other nodes, it does not verify this property and may not stabilize.

Given the collision probability $p_c = 1 - (1 - 2^{-16})^{4\overline{n_v}}$ and the probability of finding a 2-hop unique identifier $p_s = 1 - p_c$, the probability of collision after k independent trials is $P_c(k) = p_c^k$ and the probability of success is $P_s(k) = 1 - p_c^k$. Thus if we take $\delta = P_s(k-1)$, then $P_s(k) > \delta$ provided that $p_c < 1$. Notice that after only three trials the collision probability on the order of magnitude of ~ -10 for networks with a neighborhood density from 8 to 16 nodes.

3.1 Broadcast problems

One of the previously described problems of the protocol is that it relies on broadcast messages. Broadcast messages are inherently unreliable because whenever the number of nodes in the neighborhood is not known, the emitter will not be able to know if messages have arrived or not. However, in WSNs, the problem is even worse because messages may not arrive for many more reasons than in other network scenarios:

- The well known hidden terminal problem in radio networks may prevent messages from arriving without being noticed by the emitter.
- Depending on the MAC protocol, nodes may have the receiver asleep, to prevent energy loss, when a broadcast message arrives.

The common solution to improve broadcast reliability is to repeat each broadcast message several times to improve the probability of being received. However, this solution increases the potential of collision whenever several nodes are trying to broadcast a message. When some of these messages are rebroadcasts of previously arrived broadcast messages, we may be faced with the so called broadcast storm problem[17].

To reduce the broadcast storm problem we use the counter-based solution proposed in [17] enriched with distance information. In the original counter-based solution some nodes are prevented from rebroadcasting a received message in order to minimize the number of messages sent. Whenever a node receives several replicas of the same message, it concludes that most of its neighbors have already received the message, thus it does not need to send it again. By avoiding sending messages nodes are minimizing the broadcast storm problem and are saving energy but they are increasing the probability of not reaching nodes that they should. In [17], it is shown that, in a homogenous radio network, the uncovered area of a rebroadcast is directly related to the number of copies already received. In the original implementation, nodes rebroadcast after a random delay, provided that in the meantime they have not received enough copies of the same message. In the proposed solution, nodes further away from the source broadcast first, thus increasing the probability that nodes closer to the source are prevented from broadcasting.

There are other methods to minimize broadcast storms with better efficiency ratios, i.e. the ratio between the covered area and the number of broadcasting nodes is better with other methods. However, all these methods require either the knowledge of the topological localization of each node [17] or, at least, each node's neighbors [18].

In the proposed protocol, after receiving a query message, the node checks if that message has been previously received. If the message has been previously received more than a specified number of times, the message is marked as transmitted. Otherwise the message is scheduled for broadcast after a delay directly proportional to the power of the received message (Function `timetosend(int strength)` in Listing 1.2). The result is that the retransmission area is divided into concentric rings. The nodes in each of these rings rebroadcast at more or less the same time. Notice that rings are not evenly distributed in space because the reception power varies with the inverse square of the radius, which is more or less consistent with the error in measuring message strength, which is much bigger for low power receptions, i.e. outer rings are wider than inner rings because outer nodes have less accurate positioning than inner nodes.

The first question that arises is the number of copies that need to be received in order to prevent the message to be rebroadcasted. Williams and Cram [19] found that for networks with densities lower than 11 neighbors this threshold must be ≥ 4 to get a maximum coverage, i.e. minimize the number of nodes that never receive the message. However, their scenario is different from our own (we need to cover a 2 hop region while they need to cover the whole network) and they do not use the reception signal strength to schedule rebroadcasts.

The graph in Figure 3 shows the impact on the percentage of uncovered area with the chosen threshold. As expected, the uncovered area decreases with the increase of the threshold. However, it can be seen that the threshold required to achieve a significant coverage is much lower with the signal strength information than without it. To get a coverage of 99.5% (i.e. 0.5% of messages not received) we need a threshold of 6 without reception power information and a threshold of 4 with reception power information.

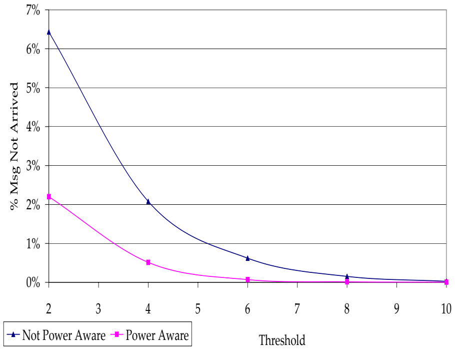

Fig. 3. Impact of the threshold value on the percentage of messages not delivered, with and without power aware rebroadcast delay.

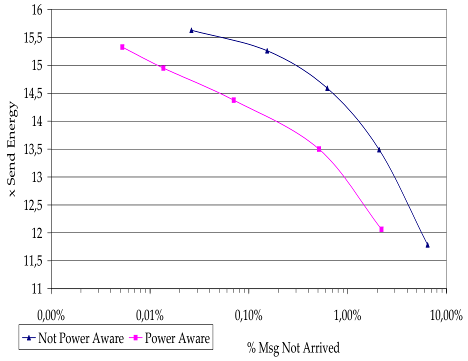

Fig. 4. Energy spent by each node (divided by the energy spent by a single message transmission) for a given coverage.

A lower threshold is better because it reduces the number of messages sent thus improving energy consumption and minimizing the broadcast storm problem. In the end, the choice is between energy and coverage. Figure 4 shows the energy spent by each node as a function of the desired coverage. In this graph, we have assumed a simplified energy model in which sending a message consumes one energy unit, the reception of a message consumes 1/10 of a unit and everything else is negligible.

Again, as expected the coverage increases with energy consumption both in the original solution and in the improved one. However, the solution which makes use of reception strength information is able to achieve better coverage with the same energy. In some cases, the uncovered area is 10 times smaller with the same energy consumption.

4 Avoiding intruders

The previous scenario assumes that every node behaves well. If one or several nodes start replying to every query saying that they have already chosen that ID, the well behaved nodes may end up with a depleted battery after repeating the query several times. If well behaved nodes do not share individual cryptographic key material with every neighbor, they are not able to distinguish well behaved neighbors from badly behaved neighbors. In such scenario, the only solution is to speak progressively softly until the badly behaved nodes are not able to hear the query. This is similar to whispering to your neighbor to prevent intruders from overhearing.

Whispering prevents nodes from communicating with more distant nodes which may have a negative impact on the network connectivity. We minimize this impact by reducing the power only as much as necessary and only in the nodes which are direct neighbors of the badly behaved one. Notice that, if a node

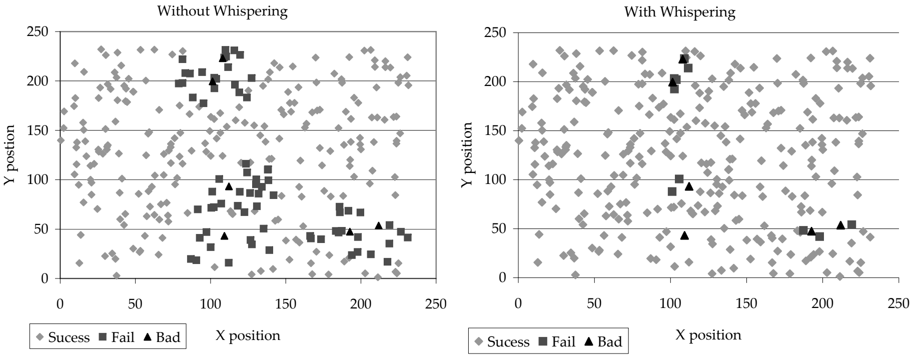

Fig. 5. Impact of whispering over the percentage of affected nodes, in the presence of a percentage of badly behaved ones.

receives a NACK originated at his neighbor's neighbor, reducing the power may prevent it from communicating with legitimate neighbors which are closer to it than the badly behaved node. It is its neighbor that should reduce transmission power. However, a node can not know for sure if a NACK is being relayed or produced at its neighbor, since a badly behaved node may always forge a NACK as if it were being relayed. Our solution was to reduce the power more quickly at nodes receiving NACKs to be relayed. Therefore, the only way a malicious node is able to force another node to reduce its transmission power rapidly is by being near, otherwise it can only affect the node through relayed NACKs.

The reduction of transmission power should only affect queries, the reply messages should be transmitted at full power, otherwise a node could be prevented from sending a NACK only because it has a badly behaved node near it.

After receiving a first query from a node, a badly behaved node may start issuing NACKs to random IDs, even if it does not receive any more queries (because of query power reduction) trying to guess the next chosen ID. To prevent it, a node should change its extend identifier every time it reduces its query transmission power.

This protocol is not able to completely prevent badly behaved nodes from stopping some well behaved nodes to choose an ID, but it minimizes the number of affected nodes. Figure 5 shows the effect of a small percentage of malicious nodes (2%) over a field of 300 randomly deployed nodes. Dark triangles represent malicious nodes, light rhombus represent nodes that were able to choose a collision free ID, and dark squares represent nodes that were not able to choose an ID, or if, with the effect of power reduction, became isolated from non-malicious nodes. As expected, the number of nodes which were not able to get an ID with whispering is much smaller than without it. With whispering, the affected nodes are in the direct vicinity of the malicious nodes, while without whispering the affected nodes are spread over their 2-hop neighborhood.

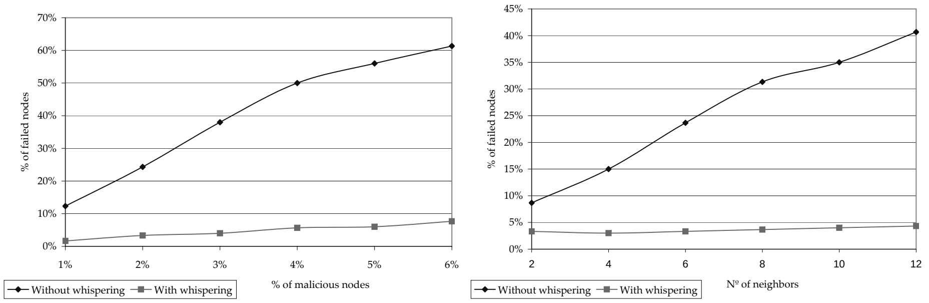

Fig. 6. Relation between the percentage of failed nodes with the percentage of malicious nodes and network density, with and without whispering

The number of affected nodes is obviously dependent on the number of badly behaved ones, but it is also dependent on the network density. The number of failed nodes increases when the number of nodes in the vicinity of malicious ones increases. Figure 6 shows how the percentage of affected nodes increases with the percentage of malicious ones and with the network density. In both cases, the percentage of failed nodes is much lower and increases much slower with whispering than without whispering. In fact, with whispering, the variation of failed nodes with the network density is almost negligible, while without whispering the effect is very noticeable.

5 Handling incrementally deployed scenarios

An important feature of address assignment protocols which is often forgotten is their ability to handle late deployed sensors and the fusion of network partitions. The deployment of additional sensors may be necessary either to improve the sensor coverage or to improve the network lifetime, in a scenario in which the sensors in place are with a nearly flat battery. The fusion of network partitions may happen either because there was an obstacle dividing nodes at the time of deployment which is now removed, or because the addition of new nodes made two or more networks reachable to each other.

In such scenarios, address collisions may happen because at the time of address assignment not every node new about each other. Most address assignment protocols do not consider these scenarios and the ones that do, choose to rerun the assignment protocol in the colliding nodes [4]. This strategy has two problems. First, it may have a negative impact on routing, because every route established through those nodes needs to be rebuilt. Second, an intruder may prevent a node from getting its ID just by sending messages with that ID, turning the whispering technique ineffective.

Another problem that these protocols need to handle is how to detect the existence of colliding addresses. In [4], address collisions are detected during the periodical neighborhood querying, which is done for this purpose. However, given that the addition of new nodes and the merging of networks are rare, such a scheme is too energy consuming. In [20] (a protocol designed for MANETs) each packet has an additional 64 bit unique number which is used to detect address collisions, but that it is not an option in WSNs given the size of each packet.

5.1 Detecting address collisions

Our approach to detect address collisions is motivated by the way that people distinguish two voices in a crowd. If one of the voices is loud and the other is soft then there are probably two persons talking. If the heard sentences do not make sense because they seem garbled, then it is possible that they are produced by more than one person. Neither of these heuristics gives precise information about the existence of colliding addresses but they may be used as triggers for a collision solving protocol.

We have added two functions to detected collisions. One should be called whenever a message arrives and another should be called whenever an out-of-order (OoO) message arrives. The first function (`recMsgPwr` in Listing 1.2) records the reception power of the last message received from each node. If the reception power changes up and down more than 10%, the collision solving protocol is started (`startColSolv` function in Listing 1.1). The second function (`recMsgOoO` in Listing 1.2) counts the number of OoO messages, and starts the collision solving protocol if the counter exceeds a specific threshold.

5.2 Collision solving protocol

The collision detection protocol described in the previous section does not provide a definitive answer about the existence of address collisions. It ends by sending a query to every node with a specific ID. Only if several nodes reply (with different extended IDs) the collision is confirmed.

When a node detects a collision it assigns a nickname to every node with the same ID, and informs the node of that nickname. The situation is similar to having two students in the same class named John, and refer to one as “Little John” and to the other as “Big John”. Notice that they will still be named John for every one else, and we cannot just name them “little” and “big”, because we would create other collisions. Moreover, in other classes they may be called by another nickname.

The solution is to reserve two bits from the 16 bit addresses for nicknames. Therefore, only 14 bit of the 16 bit addresses are assigned by the address assignment protocol (the probability of collisions increases but that is not a problem, because our address assignment protocol knows how to handle collisions), the remaining 2 bits are originally set to zero. When a node detects a collision, it informs each of the colliding nodes that their address will have some of those bits set to one. Each of the colliding nodes stores the nickname by which it is called by that node in a table. Notice that nicknames are private to their emitters, e.g. a person may be known as “Little John” by someone and “Big John” by someone else.

The collision solving process starts by broadcasting a query to every direct neighbor. Each neighbor replies with its short and extended IDs (`ColReply` message in Listing 1.1). The received extended IDs are stored in a table indexed by the corresponding short IDs. If there are no collisions the table will be just a collection of ID pairs, otherwise some short IDs will be associated with several extended IDs. After a timeout, each of the entries in the table, with more than one extended ID, is transformed into a message which is sent to every neighbor with that short ID. After receiving that message, each node takes the alias from which it becomes known by the sending node from the position of its extended ID in the message, i.e. if its extended ID is the first one in the list, it has no nickname (its nickname is 0), otherwise it adopts nicknames 1,2 or 3.

Private nicknames are very useful to handle rare collisions in already established networks because there is no ID renewed propagation. A colliding node may get different nicknames from different neighbors without having to conduct a distributed agreement between them. However, its use should be restricted to rare situations (e.g. joining of network partitions) because it takes memory to keep the nicknames and makes broadcasting harder. Since the source ID on each message depends on the destination ID, broadcasts can not be sent with a single source ID. A simple solution would be to send several broadcasts with several source IDs, but it may not work for some applications.

6 Conclusion

The address self-assigning problem is a well-studied problem in the MANET world but it has not received much attention in the WSN world. In this paper, we have described a simple address self-assignment protocol and proved its correctness. To improve the protocol performance, we have proposed an improvement to a well-known method of controlling message floods, based on the level of the power of message reception.

We have introduced the *whispering* technique to handle intruders when cryptographic keys are not available or have been compromised, and show how to use it in the proposed protocol. We believe that this is a valid security technique and intend to study its application in other protocols. Finally, we have proposed the use of *private aliases* to handle late address collisions which result from the late deployment of nodes and the fusion of network partitions.

References

1. Marrn, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., Rothermel, K.: Flexcup: A flexible and efficient code update mechanism for sensor networks. In: EWSN. (2006) 212–227
2. Dunkels, A., Grnvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: LCN. (2004) 455–462

3. IEEE: IEEE Std 802.15.4: Wireless MAC and PHY Specifications for LR-WPAN. IEEE Computer Society (may 2003)
4. Schurgers, C., Kulkarni, G., Srivastava, M.B.: Distributed assignment of encoded MAC addresses in sensor networks. In: MobiHoc, ACM (2001) 295–298
5. Elson, J., Estrin, D.: Random, ephemeral transaction identifiers in dynamic sensor networks. In: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-01), Los Alamitos, CA, IEEE Computer Society (April 16–19 2001) 459–468
6. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J.S., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Netw. **11**(1) (2003) 2–16
7. Ad, M., Perkins, C.E., Das, S.R.: IP address autoconfiguration for ad hoc networks. Internet Draft draft-ietfmanet-autoconf-01.txt, Internet Engineering Task Force, MANET WG (July 2000)
8. Gradinariu, M., Johnen, C.: Self-stabilizing neighborhood unique naming under unfair scheduler. In: Euro-Par. (2001) 458–465
9. Herman, T., Tixeuil, S.: A distributed tdma slot assignment algorithm for wireless sensor networks. In: ALGOSENSORS. (2004) 45–58
10. Nakano, K., Olariu, S.: Randomized initialization protocols for ad hoc networks. IEEE Transactions on Parallel and Distributed Systems **11**(7) (2000) 749–759
11. Angluin, D., Aspnes, J., Fischer, M.J.: Self-stabilizing population protocols. In: Ninth International Conference on Principles of Distributed Systems. (December 2005) 79–90
12. Mcauley, A.J., Manousakis, K.: Self-configuring networks. In: MILCOM 2000. 21st Century Military Communications Conference Proceedings, Los Angeles, CA, IEEE Computer Society (October 12–25 2000) 459–468
13. Gairing, M., Goddard, W., Hedetniemi, S.T., Kristiansen, P., McRae, A.A.: Distance-two information in self-stabilizing algorithms. Parallel Processing Letters **14**(3-4) (2004) 387–398
14. Moscibroda, T., Wattenhofer, R.: Coloring unstructured radio networks. In: SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2005) 39–48
15. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11) (1974) 643–644
16. Beauquier, J., Gradinariu, M., Johnen, C.: Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Technical Report 99-1225, Universite Paris Sud (1999)
17. Tseng, Y.C., Ni, S.Y., Chen, Y.S., Sheu, J.P.: The broadcast storm problem in a mobile ad hoc network. Wireless Networks **8**(2-3) (2002) 153–167
18. Lim, H., Kim, C.: Multicast tree construction and flooding in wireless ad hoc networks. In: MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems, New York, NY, USA, ACM Press (2000) 61–68
19. Williams, B., Camp, T.: Comparison of broadcasting techniques for mobile ad hoc networks. In: MobiHoc. (2002) 194–205
20. Vaidya, N.H.: Weak duplicate address detection in mobile ad hoc networks. In: MobiHoc, ACM (2002) 206–216

A Listings

```
typedef enum {
  Query, NACK, ColQuery, ColReply, ColSolve
} msgtype_t;                                   // Message Types.
typedef struct {                               // Message structure.
  uint16_t  d_id, s_id;                        // Destination and source IDs.
  msgtype_t type;                              // Message type.
  byte      hop;                               // First or second hop.
  uint64_t  xid[4];                            // Vector with extend ids.
} msg_t;
uint16_t myId, secCtr, queryPower, replyPower, pwrStep;
uint64_t myXId, prev_xid;
init() {
  myXId= largerandom();                       // Choose a random extend id
  queryPower = replyPower= MaxPower;          // Sets the power step
  pwrStep = (MaxPower-MinPower)/NCircles;     // for each ring
  newId();                                    // Chooses a new Id
}
void receiveMsg(message_t msg) {
  switch(msg.type) {
    case Query:                     // Query message received
      if(msg.s_id== myId &&
         msg.xid[0]!= myXId) {      // Test if there is a collision
        sendNack(msg);              // Send a NACK if there is a collision
      } else if(!duplicate(msg)){// Test if a copy was previously received.
         if( msg.hop == 0) {        // If it is a fist hop query schedule
            msg.hop = 1;            // a message for transmission after some
            sendQueryAfter(msg, delayStep*msg.strength/pwrStep) // time.
         }
      } else if(incCtr(msg) > MaxCtr)// Inc. and test the $n^{o}$ of copies
            markAsTrans(msg);               // If bigger than threshold, remove
      break;                                // the message from the sending queue.
    case NACK:                              // Negative ACK received
      if(msg.hop==1 &&                      // If it is a $2^{nd}$ hop message and
         (msg.xid[1]== myXId || msg.xid[1]== prev_xid)) { // it was sent
        msg.hop = 0; sendNack(msg);         // by me then send a NACK to the
      }                                     // original query node.
      if(msg.xid[msg.hop]== myXId)          // If the message was sent by me
        if(add2Ctr(msg.hop) ||              // Inc. the counter of NACKs and if
          (msg.hop==0 && msg.s_id== myId)) // exceeds the threshold or
            newId();                        // I'm the original query node,
      break;                                // choose a different Id.
    case ColQuery:                          // Receive a Collision Query message
      msg = {msg.s_id,myAddTo(msg.s_id,ColReply,myXId}; // Send a reply
      send(msg,replyPower);                 // with my short and extended IDs.
      break;
    case ColReply:                          // Receive a Collision Reply message
      addXidTo(Storage, msg.s_id, msg.xid[0]); // Add the extended ID to
      break;                                // a list indexed by the short ID.
    case ColSolve:       // Receive a Message with an ordered list of XIDs
      for(i=0; i<4 && msg.xid[i]!=myXId ; i++); // The order of each XId
      if(i>0 && i<4) addToTable(msg.s_id, i);   // is its alias. Insert
      break;                          //  the alias together with the source ID.
} }
void startColSolv(uint16_t id) {  // Start the collision solving process
  msg_t msg = {id, myAddTo(id), ColQuery}; // by sending a collision
  send(msg, queryPower);                   // solving Query,
  scheduleAlarm(onTimeout, delay);         // and schedule an alarm.
}
void onTimeout() {                        // After the alarm expires, a
  while(Storage.hasColisions()) {         // msg is sent for each ID with
    msg = Storage[i].getNextColMsg();     // colliding extended IDs.
    msg.type = ColSolve;                  // Each message contains all
    msg.s_id = myAddTo(msg.d_id);         // extended IDs associated
    send(msg,replyPower);                 // with each that ID.
  }}
```

Listing 1.1. Main protocol functions. The init() and receive() functions are the main protocol functions. The init() function just schedule a message to be sent querying for an ID. The receive() function reacts according with the messages received.

```
void newId() {
    myId = random();                    // Generate a new random short ID.
    sendQueryAfter(msg,randDelay);      // Sends a Query for that short ID.
}
boolean add2Counter(byte hop) {// Count the number of NACKs received.
  secCtr += hop==0?1:3;           // A NACK at the 2nd hop decreases
                                  // power faster than at 1st.
  if(secCtr>9 && queryPower>0) { // If to many NACKs were received,
    queryPower -= pwrStep;        // decrease the power unless power
    secCtr = 0;                   // is already zero. Reset the counter.
    prev_xid = myXId;             // Saves the previous extended ID,
    myXId= largerandom();         // and generates a new one.
    return true;                  // Inform that a new short ID must
  }                               // be generated.
  return false;                   // There is no need to generate
}                                 // a new short ID yet.
void sendQueryAfter(msg_t msg, int delay) { // Query messages are sent
    msg.type = Query; msg.xid[msg.hop] = myXid; // with queryPower,
    sendAt(msg,queryPower, time()+delay); // and with the extended ID.
}
void sendNack(msg_t msg) {
    msg.type = NACK; send(msg,replyPower); // NACK messages are sent
}                                          // with replypower.
```

Listing 1.2. Auxiliar functions.

```
typedef struct { // Nodes structure to keep out-of-order counter, last
  uint_16 id, pwr, OoOCtr; // power level, and last power level change.
  enum {up, down, level} prev;
} NodeRecord;

void recMsgPwr(int nId, int pwr) { // Message from ID with power pwr.
  NodeRecord nrec = getAddRecord(nId);// Get the record for that ID.
  if(!nrec.solving && nrec.pwr != 0) {// If it is not solving collisions
    if(nrec.pwr > 1.1*pwr) {           // if power is 10\% below last power
      if(nrec.prev == up) {            // if last change was different
        startColSolv(nId);             // starts solving collisions,
        nrec.prev = level;             // and sets the last change to none.
      } else
        nrec.prev = up;              // Otherwise sets last change to up.
    } else if(nrec.pwr < 0.9*pwr) { // If power is 10\% above last power.
      if(nrec.prevChange == down) { // if last change was different
        startColSolv(nId);          // starts solving collisions,
        nrec.prev = level;          // and sets the last change to none.
      } else
        nrec.prev = down;           // Otherwise sets last change to down.
    }
    nrec.pwr = pwr;                 // Saves the last seen power.
  }
}
void recMsgOoO(uint_16 nId) {          // Records out-of-order messages
  NodeRecord nrec = getAddRecord(nId); // Get the record for that ID.
  nrec.OoOCounter ++;               // Increments the out-of-order counter.
  if(!nrec.solving && nrec.OoOCtr > OoOThresh) { // If above threshold
    startColSolv();                    // starts the solving collisions.
    nrec.OoOCounter = 0;
  }
}
uint_16 myAddTo(uint_16 nId) {      // After the ID assignment process,
  int nickname = getFromTable(nId); // the ID of each node is specific
  return myId | nickname;           // for each correspondent node.
}
```

Listing 1.3. Collision detection functions.