Self-Organizing Sensor Networks

Doina Bein and Ajoy K. Datta

School of Computer Science, University of Nevada Las Vegas, NV 89154
{siona,datta}@cs.unlv.edu

Abstract. We propose a self-organizing protocol in a sensor network. The algorithm starting from an arbitrary state establishes a reliable communication (based on the directed diffusion strategy) in the network in finite number of steps. In directed diffusion protocol [1], a request for data from a an initiator is broadcast in the network, and the positive answers from the sensors are forwarded back to the initiator.

Keywords: Directed diffusion, routing, self-organizing, self-stabilization, sensor networks

1 Introduction

The process of sensing, data processing, and information communication is the basis of sensor networks ([2], [3]). Due to the large number of nodes and thus, the amount of overhead, the sensor nodes may not have any global identification (ID). In some cases, they may carry a global positioning system (GPS). Sensor nodes are equipped with a processor, but they have limited memory. They can carry out simple tasks and perform simple computations. The communication is wireless: radio, infrared, or optical media, and the chosen transmission medium must be available worldwide. Recent developments in wireless communications have produced low-power, low-cost, and multifunctional sensor nodes which can communicate with each other unhindered within small distances.

The nodes in sensor networks are usually deployed for specific tasks: surveillance, reconnaissance, disaster relief operations, medical assistance, etc. Increasing computing and wireless communication capabilities will expand the role of the sensors from mere information dissemination to more demanding tasks as sensor fusion, classification, collaborative target tracking. They may be deployed in an hostile environment, inaccessible terrains, and through a cooperative effort, proper information has to be passed to a higher level. Their positions are not predetermined, i.e., the network can start in an arbitrary topology.

Contributions. The goal of this paper is to design a self-organizing sensor network using self-stabilization. Both the sensors and the sensor network infrastructure are prone to failures, insufficient power supply, high error rate, disconnection, and little or no network support. Many protocols and algorithms have been proposed for traditional wireless ad-hoc networks, but they do not take into consideration frequent topology changes, sensors failures, and possible non-existent global ID. A distributed self-configuring and self-healing algorithm for

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 1233–1240, 2004.

multi-hop wireless networks is proposed in [4]. Being self-stabilizing guarantees that the system will converge to the intended behavior in finite time, regardless of the system starting state (initial state of the sensor nodes and the initial messages on the links).

In this paper, we deal with the communication reliability of the network, and we present a self-organizing protocol in a sensor network. The protocol constructs in finite number of steps a reliable communication in the sensor network in which requests for data sensed by the sensors are answered back to the initiator on the shortest path, which can send it to a higher level network (e.g. Internet, satellite etc) or application level.

Related Work. Given a general asynchronous network with at most n nodes, a similar self-stabilizing protocol is proposed in [5]. Its idea of maintaining the correct distances in the routing table regarding the closest nodes can be used for our algorithm in a sensor network, where the criterion is not the shortest distance among neighbors, but correct collected data for each request.

We use directed diffusion protocol ([1]) for implementing all the distributed services and for retrieving data through dynamically changing ad-hoc sensor networks. In directed diffusion (see Subsection 2.1), the nodes are not addressed by their IP addresses but by the data they generate. In order to be able to distinguish between neighbors, nodes may have local unique IDs. Examples of such identifiers are 802.11 MAC addresses ([6]), Bluetooth cluster addresses ([7]).

The most general technique of designing a system to tolerate arbitrary transient faults is self-stabilization ([8]). A *self-stabilizing* system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the nodes and the initial messages on the links ([9]). In a distributed self-organizing protocol, with no initialization code and with only local information, the global objective to be achieved is to construct a reliable communication in the network in which requests for data sensed by the sensors are answered back through the shortest path with the correct existing data up to the current moment, meanwhile taking care of topology changes as well.

Paper Organization. In Section 2 we present several aspects regarding sensor networks, directed diffusion protocol, and self-stabilization as a particular case of fault tolerance. In Section 3 we present the self-stabilizing directed diffusion protocol, and we make some concluding remarks and future work in Section 4.

2 Sensor Networks as Distributed Systems

A sensor node is made up of four basic components: a *sensing unit*, a *processing unit*, a *power unit*, and a *transceiver unit*. Additional components as *power generator*, *global positioning system*, *location finding unit* can be also attached, depending on the application. The *processing unit*, equipped with a small memory, manages the work of other components, it is responsible for information communication, and also supervises an eventual collaboration of the node with other nodes in accomplishing assigned sensing tasks. So it is important for our

routing algorithm to have a low computation time and memory requirements. The *transceiver unit* connects the node to the network.

The sensor nodes are generally scattered in a *sensor field*, and there are one or more nodes called *initiators*, capable to communicate with higher level networks (Internet, satellite etc) or applications. Each sensor is responsible for collecting data appropriate to its type and specifications, and maintaining a shortest path communication with the initiator nodes.

Whenever a new data is acquired regarding some past request not expired yet, a sensor node communicates it to the initiator, and facilitates the shortest path forwarding. A sensor node is able to do these tasks concurrently for any number of initiators and any number of request per initiator. Some sensor nodes may fail due to hard conditions, lack of power, environmental interference. The failure of a node should not obstruct the network functionality, and its overall task, as long as the network still remains connected and does not get disconnected.

2.1 Directed Diffusion Protocol

Directed diffusion ([1]) is a data-centric protocol, where the nodes are not identified by their ID, but by the data they generate as result of detecting (sensing). The data is organized as *attribute-value* pairs. An initiator node makes a request for a certain data by broadcasting an *interest* for a certain data throughout the sensor network. Different nodes may matched the request on different degrees, and gradients are kept in order to point out the neighbor (or the neighbors) toward the initiator (or initiators) of that particular request. The possible answers are forwarded back to the initiator, and intermediate nodes may perform a pre-filtering of the answers.

In our sensor network, we consider simple attribute-value scheme, with an upper bound of K in the number of such pairs. Each attribute has an fixed length associated value range, but not a fixed range. We impose these restrictions as the memory in a sensor unit has limited capacity. When broadcasting an interest, an initiator will specify the exact attribute name, and a much larger interval attribute. The message contains also a distance field (initially 0), and some control information: a timestamp, and an expiration value, which added to the timestamp specifies the moment in the future when the data is not important anymore (it has expired).

Every node maintains a so called *interest cache*, that contains for each existent *attribute-values* pair, a *gradient* field, which specifies the neighboring node through which an **interest** request has been received, the newest *match* up to the current moment, some control fields (*timestamp* and *expiration*), a *newest* field (identifies the node from which the newest matching data has been received) and a *distance* field (keeps track of the shortest distance toward *initiator*). An attribute exists in the cache if, at some not far moment in the past, an initiator has manifested an interest for that attribute, so the content of the cache is *interest driven*. The variables used are:

- IC = the interest cache data structure, which keeps all the data
- *no_entries_IC*: the current number of entries in the IC

- *sensed*: it is set to *true* by the sensing unit whenever new data is collected
- SU: the sensing unit data structure which contains the complete collected data, and it has the following fields: *attr* (attribute name), *value* (value of the attribute detected by the sensing unit), and *other* (other data). The sensing unit sets the Boolean variable *sensed* to *true* whenever new data is collected, and stores in the data structure SU the complete data.

There are three macros that operate over the variable IC:
- macro $add\_to\_IC(msg, nbr)$ adds a new entry based on the field values of the message msg of type $INTEREST$, sets the gradient to be nbr, and also increments $no\_entries\_IC$. If K (the upper bound in the number of entries) is reached then the oldest message, breaking ties by expiration date, is removed.
- macro $modify\_IC(e, msg, nbr)$ modifies the entry e of IC based on the field values of the message msg of type $INTEREST$, and sets the gradient to be nbr
- macro $delete\_from\_IC(e)$ deletes the entry e, and decrements $no\_entries\_IC$.

The *interest cache* is comparable with a routing cache, where instead of node IDs, we keep attribute-values pairs. Whenever an interest request is received, the node checks to see if it has an entry in its interest cache. If not, it creates an entry, using the parameters received in the message. For any node, an initiator is identified by a *gradient* field, which specifies the neighboring node through which the request has been received. The newest match in the data for an interest is kept, together with the node local ID (in the field *newest*) from which this match comes. The shortest distance toward the initiator is kept in the *distance* field. If a node receives data from its sensing unit or from the neighbors which have no existent interest, the data is simply dropped. Also, if the data received is older than the existent one, it is also dropped.

We prevent the case in which, as a result of an arbitrary initialization, for an interest received, wrong matches are stored initially in the cache with good (newer) timestamps. Whenever a new data corresponding to some pair attribute-values is received by a node (which can be either from the node itself sensing unit or from one of its neighbors) and that node ID is stored in the interest cache as the *newest*, that data will override the entry, independent of the values stored currently, even if the new value does not have a better timestamp for that interest. Later on the values received from the node sensing unit or from its neighbors will correct this.

Because the node has to self-organize, the interest cache is checked periodically for expired interests, and those entries are zapped. It is not necessary to send messages to the neighbors to do the same, because they already have the expiration date stored in their own cache, so they can do the same. Also, periodically, the entries in the cache are compared with the one of the gradients, to make sure that fictive interests are not stored as a result of an arbitrary initialization.

In [1], the model proposed forces the initiator to periodically broadcast an interest to the rest of the network, with the purpose to maintain the robustness and reliability of the network. In our algorithm, this is not necessary, because each node is responsible for keeping track of changes in its neighborhood, so an initiator will re-broadcast its interest only if, within a certain timeout, he did not receive an answer or a proper answer back. An addition to the [1], we allow any

number of initiators and any number of requests per initiator, with the condition that we have an upper bound of K in the number of entries in the interest cache, as the memory in a sensor unit has limited capacity. Another difference from [1] is that we save space by requiring only one gradient per entry, while in [1] there is a gradient for each neighbor.

2.2 Self-Stabilizing Distributed Systems

In a sensor network, the nodes communicate by messages, with two actions: *send(message)*, and *receive(message)*. The messages have variable length and they are sent through wireless links. Our algorithm is asynchronous, so it is guaranteed to run correctly in networks with arbitrary timing guarantees. A common assumption is to bound the interval of time for transmitting a message, called *timeout*, after which the message is considered lost.

There are some assumptions we make in our algorithm: independent of the node/link failure, the network never becomes disconnected, and that we have FIFO ordering among the messages on the channel, which means that the messages are delivered in a node in the order they have been sent on the channel by a neighboring node. The messages used:
- $CHECK, CLEAR$: fields *id* (sender local ID), *attr* (attribute name), *interval* (interval of values for an interest), *time* (initiator sending time), *expir* (expiration time).
- $DATA$: fields *id* (sender local ID), *attr* (attribute name), *value* (attribute value), *other_data* (other data), *time* (sending time).
- $INTEREST$: fields *attr* (attribute name), *interval* (interval of values for the attribute), *time* (initiator sending time), *expir* (expiration time of the interest), *dist* (total length of the crossed path)

Each node v has a unique local ID, LID_v, and knows only its direct neighbors (variable N_v), so it can distinguish among its adjacent wireless links. We assume that N_v as well as the *current_time* are maintained by an underlying local topology maintenance protocol, which can modify N_v value during the lifetime of the node because of adding/crashing of neighboring sensors.

Each sensor node has a local state, identified by its current interest cache, and its variables. The global state of the system is the union of the local state of its nodes, as well as the messages on the links. The distributed program consists of a finite set of actions. Each action is uniquely identified by a label and it is part of a *guarded command*: $< label >::< guard > \rightarrow < action >$

The action can be executed only if its guard, a Boolean expression involving the variables, evaluates to *true*. An action is atomically executed: the evaluation of the guard and the execution of the corresponding action are done in one atomic step. In the *distributed daemon* system, one or more nodes execute an action, and a node may take at most one action at any moment.

A self-stabilizing system S guarantees that, starting from an arbitrary global state, it reaches a legal global state within a finite number of state transitions, and remains in a legal state unless a change occurs. In a non-self-stabilizing system, the system designer needs to enumerate the accepted kinds of faults,

such as node/link failures, and he must add special mechanisms for recovery. Ideally, a system should continue its work by correctly restoring the system state whenever a fault occurs.

3 Self-Stabilizing Routing Algorithm in Sensor Networks

The macros/functions used by the algorithm are:
- *restart* sets the variables IC, SU, $no\_entries\_IC$, $turn$, $sensed$ to their default values
- *check_entry* returns *true* if an $INTEREST$ message has a matching entry in IC on the data fields, but the control fields show a newer request, $false$ otherwise
- *check_data* returns *true* if the data collected by the sensing unit or received in a DATA emssagematches an interest entry in IC, $false$ otherwise
- *match_entry* returns *true* if the data and control fields from a CLEAR/CHECK message are matching some entry in the IC, *false* otherwise

The purpose of the algorithm is to construct a reliable communication in the sensor network in which requests for data sensed by the sensors are answered back through the shortest path with the correct existing data.

The guard 1.01 has the role to check for errors as a result of an arbitrary initialization of the network, and to keep up with topology changes in the immediate neighborhood. Periodically, each entry in the interest cache IC is checked for expiration and consistency with the gradient node. If it is not expired, the data and the control fields of the entry is sent to the gradient node (message $CHECK$), to make sure that no changes have occurred in the interest requested in the past. If a change has occur or wrong interests are stored in IC, the gradient node answers back (message $CLEAR$), and the entry is cleared out of the interest cache.

Whenever a data is collected through the sensing unit, its content is stored in the data structure SU, and the variable *sensed* is set to *true*. Regulary the data collected is checked to see if it matches a previous, and not yet expired interest. If it does, then the data is sent (message $DATA$) to the corresponding gradient. In any case, *sensed* is set back to $false$. When a $DATA$ message is received, similar tests are done. If the message matches an interest, it is forwarded to the gradient, but having in the *id* field the current node (IDs can be checked only locally, they have no meaning further.)

When an initiator manifests an **interest** for some particular data, it sends an $INTEREST$ message, with data fields (attribute name and interval of values), control fields (timestamp and expiration), and a *distance* field, initially set by the initiator to be 0. Whenever an $INTEREST$ message is received by a node from a neighbor, the length of the link to the neighbor is added to the *distance* field. Among identical $INTEREST$ messages received, the one with the smallest distance value is selected, in order for the node to be always oriented using the gradient toward the initiator node through the shortest path.

Algorithm 1 $\mathcal{SOSN}$ Self-Stabilizing Directed Diffusion Protocol

$error \equiv \neg(0 \leq turn < no\_entries\_IC) \vee \neg(IC[turn].time > current\_time) \vee (IC[turn].gradient \in N_v) \vee \neg(IC[turn].newest \in (N_v \cup LID_v))$

1.01 $error \longrightarrow restart$

1.02 **timeout** $\wedge (0 \leq turn < no\_entries\_IC) \longrightarrow$
$e \leftarrow IC[turn]$
if $(e.time + e.expir < current\_time)$ **then** delete_from_IC(e)
else send $CHECK(LID_v, e(attr, values, time, expir))$ TO $e.gradient$
$turn := (turn + 1)$ mod $no\_entries\_IC$

1.03 Upon receipt of msg $CHECK$ from neighbor $nbr \longrightarrow$
if $(msg\_id \neq nbr) \wedge (msg.id \notin N_v) \vee (msg.id \in N_v \wedge (\exists$ an entry e in IC: $match\_entry(e, msg)))$ **then** discard message msg
else send $CLEAR(LID_v, msg(attr, interval, time, expir))$ TO nbr

1.04 Upon receipt of msg $CLEAR$ from neighbor $nbr \longrightarrow$
if $(msg\_id \neq nbr) \wedge (msg.id \notin N_v) \vee (msg.id \in N_v \wedge \neg(\exists$ an entry e in IC: $match\_entry(e, msg)))$ **then** discard message msg
else $delete\_from\_IC(e)$

1.05 **timeout** $\wedge$ $sensed \longrightarrow$
if $(\exists$ an entry e in IC: $check\_data(LID_v, e, SU(attr, value), current\_time))$ **then** send $DATA(LID_v, SU(attr, value, other), current\_time)$ TO $e.gradient$
$sensed := false$

1.06 Upon receipt of msg $INTEREST$ from $nbr \longrightarrow$
if $\neg(msg.id \in N_v \wedge msg.id == nbr)$ **then** discard message msg
$msg.dist = msg.dist + length\_path\_to(nbr)$
if $(\exists$ an entry e in IC: $match\_entry(e, msg))$ **then**
if $(msg.dist < e.dist \vee nbr == e.gradient)$ **then**
$modify\_IC(e, msg, nbr)$
send msg TO all nodes in $N_v \setminus nbr$
else discard message msg
else
if $(\exists$ an entry e in IC: $check\_entry(e, msg))$ **then**
$modify\_IC(e, msg, nbr)$
else $add\_to\_IC(msg, nbr)$
send msg TO all nodes in $N_v \setminus nbr$

1.07 Upon receipt of msg $DATA$ from $nbr \longrightarrow$
if $(msg.id \in N_v \wedge msg.id == nbr) \wedge (\exists$ an entry e in IC:$check\_data(msg.id, e, msg(attr, value, time)))$ **then** send $DATA(LID_v, msg(attr, value, other, time))$ TO $e.gradient$

4 Conclusion

We presented a self-organizing protocol that guarantees that starting in an arbitrary state, and having only local information, in finite number of steps builds a reliable communication in the network based on directed diffusion method. An interesting open problem is a comparative study between self-stabilizing directed diffusion and snap-stabilization. Snap-stabilization was first introduced in [10], and guarantees that a system will always behave according to its specifications ([11]). Snap-stabilizing propagation of information with feedback algorithm has been presented in [12] and is used extensively in distributed computing to solve problems like spanning tree construction, termination detection, synchronization.

References

1. Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. *Proceedings of the 6th annual international conference on Mobile computing and networking, Boston, Massachusetts, United States*, pages 56 – 67, 2000.
2. Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramanian, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine August 2002*, pages 102–114, 2002.
3. G. Hoblos, M. Staroswiecki, and A. Aitouche. Optimal design of fault tolerant sensor networks. *IEEE International Conference Cont. Apps. Anchorage, AK*, pages 467–472, 2000.
4. H. Zhang and A. Arora. GS^3: Scalable self-configuration and self-healing in wireless networks. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.
5. D. Bein, A. K. Datta, and V. Villain. Self-stabilizing routing protocol for general networks. *Second edition of RoEduNet International Conference In Networking, Iasi, Romania*, pages 15–22, 2003.
6. IEEE Computer Society LAN MAN Standards Committee. Wireless lan medium access control (mac) and physical layer (phy) specifications. *Technical Report 802.11-1997, Institute of Electrical and Electronics Engineers New York, NY*, 1997.
7. The Bluetooth Special Interest Group. Bluetooth v1.0b specification. *http://www.bluetooth.com*, 1999.
8. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.
9. M. G. Gouda. *Elements of network protocol design.* John Wiley & Sons, Inc., 1998.
10. A. Bui, AK Datta, F Petit, and V Villain. Space optimal snap-stabilizing pif in tree networks. *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, 1999.
11. A Cournier, AK Datta, F Petit, and V Villain. Optimal snap-stabilizing pif in unoriented trees. *5th International Conference On Principles Of Distributed Systems (OPODIS 2001)*, pages 71–90, 2001.
12. A Cournier, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in arbitrary networks. In *IEEE 22nd International Conference on Distributed Computing Systems (ICDCS 02)*, pages 199–206. IEEE Computer Society Press, 2002.