Generic Role Assignment for Wireless Sensor Networks*

Kay Römer, Christian Frank
Department of Computer Science
ETH Zurich, Switzerland
{roemer,chfrank}@inf.ethz.ch

Pedro José Marrón, Christian Becker
Department of Computer Science
University of Stuttgart, Germany
{marron,becker}@informatik.uni-stuttgart.de

Abstract. Wireless ad hoc networks of sensor nodes are envisioned to be deployed in the physical environment to monitor a wide variety of real-world phenomena. Almost any sensor network application requires some form of self-configuration, where sensor nodes take on specific functions or roles in the network without manual intervention. These roles may be based on varying sensor node properties (e.g., available sensors, location, network neighbors) and may be used to support applications requiring heterogeneous node functionality (e.g., clustering, data aggregation). In this paper we argue that the assignment of user-defined roles is a fundamental part of a wide range of sensor network applications. Consequently, a framework for assignment of roles to sensor nodes in an application-specific manner could significantly ease sensor network programming. We outline the general structure of such a framework and present a first approach to its realization. We demonstrate its utility and feasibility using a number of concrete examples.

1 Introduction

Wireless sensor networks consist of so-called sensor nodes – small untethered computing devices equipped with sensors, a wireless radio, a processor, and autonomous power supply. Large and dense networks of these devices can then be deployed unobtrusively in the physical environment in order to monitor a wide variety of real-world phenomena with unprecedented quality and scale while only marginally disturbing the observed physical processes [2].

Many sensor network applications require some form of configuration, where sensor nodes take on specific functions in the network. Configuration of a sensor network is particularly challenging, as the anticipated large scale of sensor networks (in terms of numbers of nodes) typically precludes manual configuration of individual nodes. Additionally, pre-deployment configuration is often infeasible because some configuration parameters such as node location and network neighborhood are typically unknown prior to deployment. Also, node parameters may change over time, necessitating dynamic re-configuration. Hence, a means for *in-situ* configuration after deployment must be found. The term *self-configuration* is commonly used to express the fact that a sensor network should configure itself without manual intervention.

When sensor nodes join the network, they are in an initial software state. However, nodes may differ in their hardware capabilities and parameters such as their location or their network neighborhood. The goal of configuration is to break the initial symmetry and assign specific *roles* to individual sensor nodes based on their properties. As the network and node properties change over time, role assignments must be updated to reflect these changes. Based on the assigned roles, sensor nodes may adapt their behavior accordingly, establish cooperation with other nodes, or may even download specific code for the selected role.

Examples that would require some form of self-configuration and role assignment can be readily found in the sensor network literature. Below we present three of them, which we will use to illustrate our approaches throughout the paper. Variations and combinations of these examples show up in many applications.

Coverage. A certain area is said to be covered if every physical spot falls within the observation range of at least one sensor node. In dense networks, each physical spot may be covered by many equivalent nodes. The lifetime of the sensor network can be extended by turning off these redundant nodes and by switching them on again when previously active nodes run out of battery power [13]. Essentially, this requires proper assignment of the roles `ON` and `OFF` to sensor nodes. □

Clustering. Clustering is a common technique to improve the efficiency of data delivery (e.g., flooding, routing) [6]. With clustering, one of the three roles `CLUSTERHEAD`, `GATEWAY`, `SLAVE` is assigned to each node. A clusterhead acts as a hub for slaves in its neighborhood such that slaves directly communicate with their clusterhead only. Gateways are slaves of more than one cluster and interconnect multiple clusters by forwarding messages between them. □

In-Network Aggregation. Due to the scarcity of energy and the high energy cost of wireless communication, reducing data communication is an important design goal in

*This work was partly supported by NCCR-MICS, a center supported by the Swiss National Science Foundation under grant no. 5005-67322.

sensor networks. One common form of data reduction is *in-network data aggregation*, where certain nodes in the network aggregate sensory data from many sources [4]. For this, sensor nodes must be assigned the roles `SOURCE` (generate sensory data), `AGGREGATOR` (aggregate data), and `SINK` (consume aggregated data) roles. In order to achieve a significant network traffic reduction, aggregator nodes should be located close to the data sources they aggregate. □

As illustrated by these examples, role assignment is fundamental for self-configuration. The criteria for role assignment are manifold and can vary significantly from application to application. While previous work has proposed individual solutions for specific self-configuration problems (e.g., the ones sketched above), we aim to provide a generic framework that supports the development of self-configuring applications with programming abstractions for role assignment. We believe that such a framework may substantially facilitate sensor network programming.

However, it is not obvious whether and how role assignment can be efficiently supported by a single framework for a variety of applications. We will investigate this issue in the next sections. Section 2 defines the core elements that must be present in such a framework. Section 3 will then sketch one specific instance of a role assignment approach by formulating exemplary role assignment rules for the examples above and outlining an algorithm for generic role assignment. Section 4 discusses related work and Section 5 concludes the paper and provides an insight into future work.

2 Core Elements

From the above examples we derive the need for four core elements of any system that supports role assignment. First, a *property directory* which provides access to the capabilities and parameters of sensor nodes. Secondly, a *role specification* defines possible roles and rules for how to assign them. Thirdly, a *role assignment algorithm* assigns roles to sensor nodes taking into account role specifications and properties. Finally, a number of *basic services* may be required.

Property Directory. Properties of individual sensor nodes are available sensors (e.g., temperature) and their characteristics (e.g., resolution); other hardware features (e.g., memory size, processing power, communication bandwidth); remaining battery power; or physical location and orientation. Some properties are static, some may change over the lifetime of the network. However, we assume that properties are not subject to frequent significant changes. This reflects the understanding that a particular configuration is valid for a certain minimum amount of time. Depending on their nature, properties may be defined at production time, by hardware introspection, or by sensors. The property directory abstracts the dissimilitude of properties and provides a unified interface to access property values. There is one such directory on each sensor node, which is independent of the directories on other nodes.

Role Specification. In its basic form, a role is an identifier (e.g., `CH` for clusterhead, `GW` for gateway). We found it useful to augment roles with parameters that further refine a role. In the clustering example, gateways may be assigned a parameterized role `GW(CH1,CH2)`, where the parameters refer to the clusterheads connected by the gateway. The values of role parameters can be accessed by the application. For example, a cluster-based data routing algorithm might have to know the cluster heads a node belongs to.

A set of rules defines the necessary conditions for the assignment of roles. In general, these rules will refer to a set of sensor nodes and their respective properties. That is, the decision of which role should be assigned to a single sensor node typically depends on a set of sensor nodes. We assume that all sensor nodes are subject to the same set of roles and according rules. This reflects the understanding that all sensor nodes are in the same initial software state.

Role Assignment Algorithm. The task of this component is to assign roles to sensor nodes, taking into account role specifications and sensor node properties. Depending on the specific problem instance, it might be useful to allow the assignment of multiple roles to one node. For example, a single sensor node might act both as a data source and as an aggregator. Property changes and node failures may necessitate re-assignment of roles.

The large scale, resource and energy scarcity, and robustness requirements of sensor networks imply that role assignment algorithms should be distributed and localized algorithms, where interaction is limited to nodes in the network neighborhood. We will discuss such an algorithm in Section 3.3.

Basic Services. A number of services such as node localization, neighbor/topology discovery, or time synchronization may be needed for role assignment. However, it should be possible to reuse existing approaches for this purpose, possibly requiring minor adaptations. Hence, we do not discuss these services in more detail.

3 A Generic Role Assignment Scheme

In this section we will sketch one possible specific instance of a framework that supports role assignment. We will first give an overview of this approach. We then show how this approach can be used for a number of applications and outline a possible implementation of our role assignment algorithm that informally delineates the feasibility of our approach.

3.1 Overview

In our approach, the property directory exports property values as a list of name-value pairs. Moreover, it can provide an indication if a property value changes. The role specification is a list of role-rule pairs. For each possible role, the associated rule specifies the conditions for assigning this role. Rules are Boolean expressions that may contain predicates over the local properties of a sensor node and predicates over the properties of well-defined sets of nodes in the neighborhood of a sensor node. All nodes in the network have a copy of the same role specification.

A separate instance of the role assignment algorithm is executing on each sensor node. Triggered by property and role changes on nodes in the neighborhood, the algorithm evaluates the rules contained in the role specification. If a rule evaluates to true, the associated role is assigned.

For the ease of exposition, we do not explicitly discuss assignment of multiple roles to a single sensor node. However, there is nothing particular in our approach that prevents us from supporting this.

3.2 Application Examples

Let us now revise the examples sketched in the introduction into more formal role specifications. Note that these role specifications will typically result in approximate solutions of the respective configuration problems.

Coverage. As mentioned earlier, nodes must be assigned `ON` and `OFF` roles. Requirements for the assignment of these roles are that the area of interest is covered by the sensors of `ON` nodes, and that `ON` nodes have sufficient remaining battery power. Assuming one is interested in coverage with temperature readings, one possible formulation could be:

```
1 ON :: {
2   sensor == temp &&
3   battery >= threshold &&
4   count(2 meters) {
5     role == ON
6   } <= 1 }
7 OFF :: else
```

The rule in lines 1-6 specifies the conditions for a node to have `ON` status: it must have a temperature sensor and enough battery power (lines 2 and 3) and there must be at most one other `ON` neighbor within its sensing radius of 2 meters (lines 4-6). Otherwise the node is assigned `OFF` status.

The property directory needs to contain an entry for the current battery level `battery=<remaining energy>` and an entry `sensor=temp` for nodes that are equipped with a temperature sensor. In addition, there is an entry `role` that holds the current role of the node.

The `count` operator in line 4 expects the specification of a set of sensor nodes as its first parameter and returns the number of nodes for which the expression in curly braces evaluates to true. Note that the variables (e.g., `role` in line 5) in these expressions refer to properties of the specified neighbor nodes. The first parameter to the `count` operator may take various forms, here we motivate a metric radius; another form would be a hop-radius, which is illustrated by the following examples. The output of a sample simulation run is shown in Fig. 1(a). □

Clustering. A clustering approach needs to define assignment rules for `CLUSTERHEAD`, `GATEWAY` and `SLAVE` roles. The assignment of these roles depends on a variety of parameters. Clusterheads should be more powerful devices (in terms of processing, memory, communication, and energy supply), since they act as hubs for many slaves. This may be easily formulated in terms of the property directory and is neglected here. For the role assignment, consider the following basic scheme:

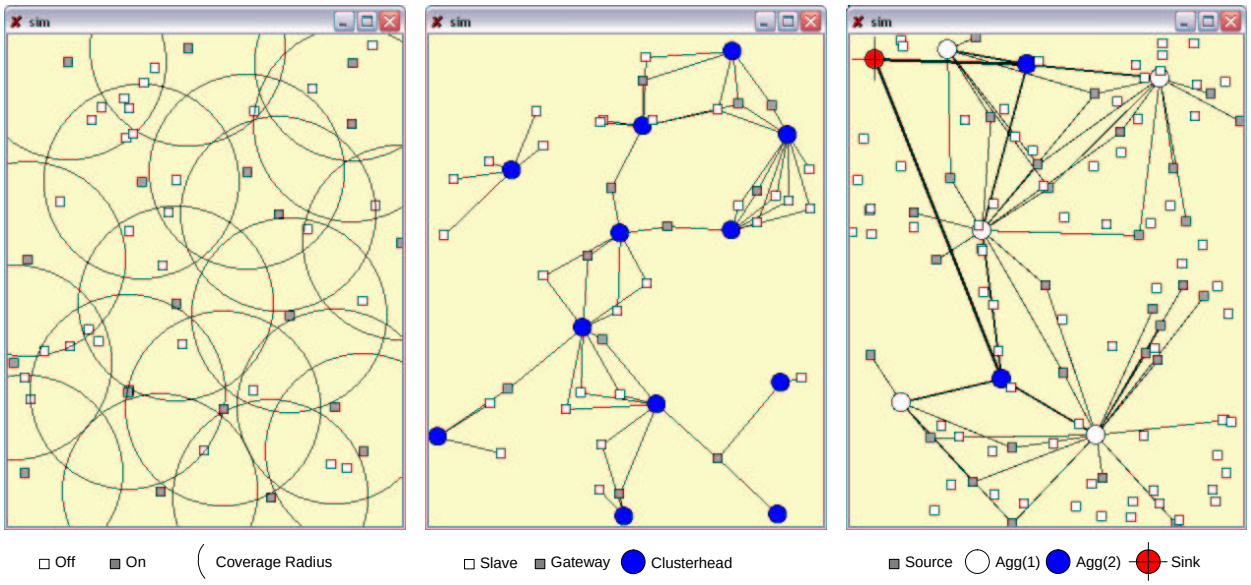
```
1 CLUSTERHEAD :: {
2   count(1 hop) {
3     role == CLUSTERHEAD
4   } == 0 }
5 GATEWAY(c1,c2) :: {
6   retrieve(1 hop, 2) {
7     role == CLUSTERHEAD
8   } == (c1,c2) &&
9   count(2 hops) {
10    role == GATEWAY(c1,c2)
11  } == 0 }
12 SLAVE :: else
```

A node that does not have any clusterhead among its neighbors declares itself `CLUSTERHEAD` (lines 1-4). Note that a one-hop radius is used for the `count` operator.

Nodes should be assigned the role `GATEWAY` if they are neighbors to at least two clusterheads but are not aware of any other gateway nodes connecting the same two clusterheads. Note that the `GATEWAY` role is additionally parameterized to the clusterheads it connects.

To achieve this we introduce the `retrieve` operator (line 6), which is similar to `count`, but returns a list of node identifiers instead of only counting the nodes. Usage of the `retrieve` operator by the programmer implies the need for locally unique node identifiers at the system level. In this example the operator is used to identify two clusterheads in the neighborhood of the node and to bind them to the names `c1` and `c2` in line 8 (similar to binding of variables in declarative programming languages). The second parameter to `retrieve` in line 6 requests any two matching nodes. If not enough matching nodes exist, the `retrieve` expression evaluates to false. In this case, the `GATEWAY` role is not assigned, the parameters are not bound, and the evaluation of lines 9-11 can be omitted.

The output of a sample simulation run is shown in Fig. 1(b). The graph includes the links between clusterheads and their slaves, demonstrating that network connectivity is maintained. □



(a) Coverage (b) Clustering (c) Aggregation

Figure 1: Sample simulation runs to qualitatively illustrate our approaches. The simulation is based on the algorithm described in Section 3.3. Nodes with a communication radius of 20m are uniformly placed in an area of 100m $\times$ 80m.

In-Network Aggregation. In this example, sensor nodes equipped with temperature sensors act as data sources. Our goal is to designate aggregator nodes in the close neighborhood of these data sources that aggregate (e.g., average) temperature readings from many data sources. A single sink node with known position consumes aggregated data. Since data flows from sources to the sink, aggregator locations should be directed towards the sink.

The role specification contains recursive rules for aggregators of different levels (while an aggregator of level 0 is a source):

```
1  AGG(0) :: { sensor == temp }
2  AGG(N) :: {
3    count(2 hops) {
4      role == AGG(N-1) &&
5      dist(pos, sink_pos) >
6        dist(super.pos, sink_pos)
7    } >= 2 &&
8    count(2 hops) {
9      role == AGG(N)
10   } == 0 &&
11   1 <= N && N <= 2 }
```

The aggregators of level 0 (sources) are defined in line 1 as nodes with a temperature sensor. We recursively define aggregators of higher level as follows: A node becomes an aggregator of level N if there are at least two data sources (aggregators of level N-1) in its 2-hop neighborhood which are farther away from the sink (lines 3-7). Additionally, there must not be other aggregator nodes of the same level in its neighborhood (lines 8-10). Only aggregator levels 1 and 2 are defined by this rule (line 11). Note that the recursive definition can be easily expanded based on the restrictions in line 11.

While the Boolean expressions for `count` and `retrieve` are generally evaluated for the remote (neighboring) nodes, these can refer to properties of the ancestor node by prepending the prefix `super` (like `super.pos` in line 6). The `dist` operator (lines 8 and 9) returns the distance between two positions. `sink_pos` refers to the position of the sink (e.g., a fixed base station), which we assume to be known.

The above aggregator roles could be augmented with a `parent` parameter pointing to an aggregator of the next higher level where applicable to support data routing. Sources and aggregators would then send data to this parent. If nodes end up without a parent (i.e., nodes that have no aggregators/sources in its 2-hop neighborhood) then they directly send data to the sink. Fig. 1(c) shows the output of a sample simulation run. The lines indicate the aggregation structure, where thicker lines denote higher aggregation levels. □

3.3 Role Assignment Algorithm

We would like to present a first straight-forward (and inefficient) algorithm that can be used to implement rule evaluation. Evaluating a rule involves evaluation of a Boolean expression, which may contain `count(rad){expr}` and `retrieve(rad){expr}` operators. To evaluate these operators in all rules in parallel, the initiator broadcasts a `request` message containing its ID and the values of any `super.*` properties to its `max(rad)` neighborhood. The receivers know the `expr` of all operators, since all nodes share the same rules. `max(rad)` refers to the maximum

radius of all operators in the role specification. If the receiver of a `request` message is also currently evaluating a rule and its ID is lower then the ID of the sender, it sends back an `abort` message to the initiator to ensure atomic rule evaluation (see below). Otherwise, the receiver evaluates the `expr`s locally and sends back a `reply` message to the initiator containing its ID and the outcome of the evaluation of the `expr`s. If the initiator receives any `abort`, it aborts the process of rule evaluation. Otherwise it uses the received `reply` messages to complete local evaluation of the rules. If the initiator has changed its role as a result of rule evaluation, if there were any `abort`s, or if local properties changed since last rule evaluation, then the initiator broadcasts a `confirm` message to the `max(rad)` neighborhood, containing its new role where applicable, and an indication of property changes where applicable. Upon receiving a `confirm` message, nodes restart rule evaluation if they were aborted earlier, if the role of the sender has changed, or if sender's properties changed. Initially, all nodes start rule evaluation. In order to reduce the probability of aborts, start of rule evaluation is randomly delayed. Property changes also trigger rule evaluation.

There are several major issues with the correctness, robustness, and efficiency of the above algorithm. The tentative approaches to these issues described below should eventually lead to an efficient distributed algorithm for rule evaluation.

Correctness. The first and most important issue is the problem of how to ensure *safety* and *liveness*, that is, in absence of property changes all nodes should eventually decide on a *stable* role that does not trigger role changes at any other node. In general, these properties are inherently tied to a particular role specification and must be ensured by the role programmer. In some cases, the developer can use simple heuristics, such as ensuring that a node may not return to an earlier role. Where this is not possible, the runtime system may detect patterns of misbehavior (e.g., a node going through the same cycle of roles over and over again) and notify the application. In such cases the application could take actions to fix this situation, for example by using the currently selected role as an approximation (which would make sense for the coverage problem, for example), or by notifying a human operator. Property changes (which are also propagated to neighbors by `confirm` messages) reset the algorithm to allow dynamic re-configuration. In order to notify the application of a selected role, it must additionally be possible to decide when a stable role has been assumed. We are currently examining an approach where a role is considered stable when it did not change for a certain amount of time.

In the above examples we implicitly assumed that rules are evaluated *atomically*: if the outcome of evaluation of the rules on node N depends on a sensor node M, then M must not change its role during the evaluation of the rules on node N. We are currently exploring the use of synchronized physical time among sensor nodes for an efficient implementation of rule atomicity. One option is to include begin and end times of rule evaluation in `confirm` messages to provide an indicator for the need of abortion. A second option is to setup a rule evaluation schedule that avoids the need for rule abortions. It might even be an option to use randomization techniques to establish schedules which ensure atomicity with high probability.

Robustness is an important issue in sensor networks, because sensor nodes and communication links are subject to frequent failures. Note that node failures which happen outside a role assignment cycle (the `request`, `reply`, `confirm` sequence described above) are considered implicitly by the above algorithm, since the failed node's properties will no longer be part of future assignment decisions. This, however, requires to trigger dynamic re-configuration for affected nodes after a node failure. This could be implemented by a failure-detection service or by periodic re-evaluation of rules.

The presented algorithm would be particularly sensitive to failures that happen during a role assignment cycle. Lost `reply` messages will cause the algorithm to ignore respective nodes in the role assignment decision, missing `confirm` messages would inhibit aborted nodes from re-evaluating rules. In both cases, timeouts can be used: to ignore late messages in the former case and to re-start evaluation on aborted nodes in the latter.

Efficiency. Of particular importance for the performance of the approach are the implementations of the `count` and `retrieve` operators. A straight-forward approach as in the above algorithm would be inefficient due to the induced message overhead. One possible solution to this problem is to exploit the fact that all nodes execute the same rules, so that a node is able to determine which nodes are affected by a role change, and what information these nodes will need in order to re-evaluate their rules. Hence, a node where a role change occurs can proactively send necessary information to the affected nodes. If affected nodes have cached information from unchanged nodes, proactive updates will suffice to re-evaluate rules on affected nodes without a need for repeated broadcasts.

One particular way to restrict broadcasts is the observation that all information needed to evaluate `count` and `retrieve` statements is locally bounded by a given number of hops and/or geographical scope. Therefore information needed to evaluate a predicate needs to be propagated only up to the maximum range parameter of its enclosing `count/retrieve` context. The *locality* of the algorithm may be directly inferred from the programmer's specification.

Since parsing and interpreting role specifications on sensor nodes might be too costly or infeasible, we are exploring ways of *pre-compiling* role specifications offline. The out-

put of pre-compilation will contain the role assignment algorithm that has been parameterized based on the role specifications. This approach should both result in compact and efficient code.

4 Related Work

There exists a large body of literature on self-organization and self-configuration in a number of fields (e.g., robotics). Due to space limitations, we cannot review these here. Self-configuration in ad hoc and sensor networks has been an active research topic in the recent past. Various approaches for solving *specific* self-configuration problems have been devised. Examples include coverage [7]; aggregator placement [3]; clustering, routing and addressing [5, 8, 9]. [5] uses a fixed set of roles to build a network-wide backbone infrastructure. However, none of these approaches are *generic* frameworks that support the assignment of user-defined roles in application-specific manner.

Only recently, neigborhood programming abstractions [10, 11] have been proposed, where network neighbors can easily share variables among each other. One possible way to implement our approach could be on top of such an abstraction.

Inspired by cellular cooperation in biological organisms, Amorphous Computing [1] explores ways to program smart matter – very densely deployed collections of indistinguishable smart particles. In contrast, our approach is based on the observation that sensor nodes may significantly differ in their properties, may rely on a number of basic services (e.g., localization), and are less densely deployed. Also, we focus on the configuration of sensor networks, the actual "programming" (i.e., distributed data processing etc.) is not part of our work, although role parameters may provide valueable input for it.

Our scheme for role assignment is similar to cellular automata [12], where the state of a particle in a regular arrangement is completely defined by the previous values of a neighborhood of particles around it. Note that a classification of a subclass of cellular automata in [12] indicates that a large group of automata converges to well-defined states. Major differences of our approach are that state updates are not synchronous, sensor nodes are not in a regular arrangement, and sensor nodes differ in their properties.

5 Conclusion and Outlook

We have presented an initial approach to solve the problem of generic role assignment by providing a practical and feasible tool for the development of sensor network applications. We have also outlined the general structure of frameworks for role assignment and presented a first approach to realize such a framework, demonstrating its utility by means of a number of concrete examples. In order to support the feasibility of this scheme, we sketched a straight-forward distributed role assignment algorithm and presented tentative approaches for a more efficient and robust implementation.

As mentioned in Section 3.2, our approach might not be able to generate optimal configurations under certain circumstances (e.g., an aggregator placement with minimum energy expenditure), but we believe that even suboptimal configurations are helpful for many applications. Moreover, the generated configurations may serve as a starting point for further local optimizations (e.g., as in [3]).

Our current work includes the development and evaluation of an efficient distributed role assignment algorithm, and the development of an encompassing language for role specifications. We plan to extend our approach for generic role assignment into a set of tools and services that support the development of self-configuring sensor network applications.

References

[1] H. Abelson et al. Amorphous Computing. *CACM*, 43(5):74–82, March 2000.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, March 2002.

[3] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, Berkeley, USA, April 2003.

[4] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *MobiCom*, Boston, USA, August 2000.

[5] M. Kochhal, L. Schwiebert, and S. Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *WSNA*, San Diego, USA, 2003.

[6] T. J. Kwon and M. Gerla. Efficient Flooding with Passive Clustering (PC) in Ad Hoc Networks. *Computer Communication Review*, 32(1):44–56, January 2002.

[7] S. Slijepcevic and M. Potkonjak. Power efficient organization of wireless sensor networks. In *ICC*, Helsinki, Finland, June 2001.

[8] K. Sohrabi, V. Ailawadhi, J. Gao, and G. Pottie. Protocols for Self Organization of a Wireless Sensor Network. *Personal Communication Magazine*, 7:16–27, 2000.

[9] L. Subramanian and R. H.Katz. An architecture for building self-configurable systems. In *MobiHoc*, Boston, USA, August 2000.

[10] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, Boston, USA, March 2004.

[11] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *MobiSys*, Boston, USA, June 2004.

[12] S. Wolfram. *Cellular Automata and Complexity*. Addison-Wesley, 1994.

[13] Y. Xu, J. Heidemann, and D. Estrin. Geography-Informed Energy Conservation for Ad-Hoc Routing. In *MobiCom*, Rome, Italy, July 2001.