TinyCubus: A Flexible and Adaptive Framework for Sensor Networks

Pedro José Marrón, Andreas Lachenmann, Daniel Minder,
Jörg Hähner, Robert Sauter, and Kurt Rothermel
University of Stuttgart, Germany
Institute of Parallel and Distributed Systems (IPVS)
{marron,lachenmann,minder,haehner,sauterrt,rothermel}@informatik.uni-stuttgart.de

Abstract— **With the proliferation of sensor networks and sensor network applications, the overall complexity of such systems is continuously increasing. Sensor networks are now heterogeneous in terms of their hardware characteristics and application requirements even within a single network. In addition, the requirements of currently supported applications are expected to change over time. All of this makes developing, deploying, and optimizing sensor network applications an extremely difficult task. In this paper, we present the architecture of `TinyCubus`, a flexible and adaptive cross-layer framework for TinyOS-based sensor networks that aims at providing the necessary infrastructure to cope with the complexity of such systems. `TinyCubus` consists of a data management framework that selects and adapts both system and data management components, a cross-layer framework that enables optimizations through cross-layer interactions, and a configuration engine that installs components dynamically. Furthermore, we show the feasibility of our architecture by describing and evaluating a code distribution algorithm that uses application knowledge about the sensor topology in order to optimize its behavior.**

I. Introduction

In the last few years wireless sensor networks have been proposed as a way to unobtrusively gather real-world data. A sensor network consists of small networked devices that are equipped with sensors. Each node is able to process data in the network and transmit it using multi-hop communication. Most nodes are resource-constrained and, additionally, for many applications energy consumption plays an important role. Finally, sensor nodes do not have to be stationary, and may even move at high speeds.

In order to acquire data, sensor networks use various kinds of hardware. Although many research groups use Berkeley Motes together with TinyOS [1], there is no standard platform for sensor nodes yet. Even different models of motes running TinyOS differ greatly. Currently, motes with new means of communication such as IEEE 802.15.4 devices are being developed, and the integration of new types of sensors, based on MEMS technology, for example, are under way.

Likewise, applications are continuously evolving and are, therefore, highly heterogeneous. New applications continue to appear and although there are similarities, each of them has its own specific requirements. For example, there are well-known applications whose goal is to monitor ecological phenomena using sensor networks [2], whereas others are developed for military operations, medical care or rescue operations.

The network itself, defined as a collection of devices, might also be heterogeneous: In more recent applications, a network often consists of different devices that are able to perform different tasks. For example, some nodes are equipped with special kinds of sensors, whereas others may have more processing power for complex calculations or act as gateways to infrastructure-based networks. Furthermore, the specific requirements for the network depend heavily on the application, as we will see in the next sections. If these requirements change or another application is executed, the network has to adapt. Developing adaptation for every application and optimizing the code over and over again are complex, error-prone tasks. In order to simplify application development, system software in the form of a flexible, adaptive framework that supports a large number of hardware platforms and applications is clearly needed.

In this paper we present the architecture of `TinyCubus`, which aims at providing the necessary infrastructure to support the complexity of such systems. `TinyCubus` consists of a data management framework, a cross-layer framework, and a configuration engine [3]. The *data management framework* allows the dynamic selection and adaptation of system and data management components. The *cross-layer framework* supports data

sharing and other forms of interaction between components in order to achieve cross-layer optimizations. The *configuration engine* allows code to be distributed reliably and efficiently by taking into account the topology of sensors and their assigned functionality.

The contribution of this paper is twofold. First, we describe the architecture of `TinyCubus`, a flexible, adaptive cross-layer framework for sensor networks. Secondly, we describe and evaluate a code distribution algorithm used by the configuration engine to disseminate components and code reliably and efficiently within the network, using the cross-layer data provided by the framework. The results of our evaluation show that our algorithm reduces the number of messages exchanged if the topology of the network is structured and known to the application.

The remainder of this paper is structured as follows. The next section describes the requirements of two specific sensor network applications. Section III presents the overall architecture of our framework and gives more detailed information about its three parts. Section IV describes and evaluates the code distribution algorithm used by the configuration engine. Section V gives an overview of related work and section VI concludes this paper and describes future directions.

II. Application Requirements

Two specific sensor network applications play an important role in the research performed at the University of Stuttgart: Sustainable Bridges [4] and Cartalk 2000 [5]. Both applications are being studied as canonical examples for a wide range of applications that deal with static and mobile sensor nodes. Their analysis allows us to identify requirements and characteristics that apply to applications that fall in this category.

The goal of the Sustainable Bridges project is to provide cost-effective monitoring of bridges using static sensor nodes in order to detect structural defects as soon as they appear. A wide range of sensor data is needed to achieve this goal, e.g., temperature, relative humidity, swing level, vibrations, as well as noise detection and localization mechanisms to determine the position of cracks. In order to perform this localization, nodes sample noise emitted by the bridge at a rate of 40 kHz and, by using triangulation methods, the position of the possible defect is determined. This process requires the clocks of adjacent sensors to be synchronized within 15-25 μs of each other. Finally, sensors are required to have a lifetime of at least 3 years so that batteries can be replaced during the regular bridge inspections.

In contrast, the goal of the Cartalk 2000 project is to develop a cooperative driver assistance system that provides an ad-hoc warning system for traffic jams, accidents, and lane or highway merging. In addition, information such as average speed, road conditions, and position can be requested through a standard query interface. Since sensors are integrated into cars, they are mobile with respect to each other. A wide range of highly dynamic sensor data, e.g., speed, position, and tire pressure, is gathered continuously. The processing of data must be performed in a timely manner and immediately sent to other drivers that might be interested in it. Thus, time-constrained communication is important for the system since data must be forwarded to the appropriate cars at the right time. In contrast to the Sustainable Bridges application, energy constraints are less severe in this application since sensor nodes are directly connected to the electric system of the car.

A. Application Similarities

As can be extracted from the description of the projects, the Sustainable Bridges and the Cartalk 2000 applications have some similarities. Both are mostly data-centric or data-driven. They are also state-based, that is, their needs might change depending on the current state of the application. Sustainable Bridges, for example, has a monitoring state in which it is most important to detect the occurence of an event and to notify other nodes as fast as possible. Having recorded data with a high sampling rate, the nodes switch to the analyzing state in which they reliably exchange and analyze the recorded data. Moreover, both applications must be fault-tolerant with respect to failures and changes in environmental conditions, since they are expected to operate unattended for long periods of time. Since the Sustainable Bridges and Cartalk 2000 applications perform sensitive monitoring tasks, both applications need to be reliable and the availability of sensors has to be guaranteed. Finally, since some of the application requirements may change over time, the software running on the sensor nodes should be able to adapt or reconfigure itself so that the right functionality can be chosen at the appropriate time.

B. Application Differences

However, these applications also have considerable differences. Table I provides an overview of the different requirements found in both applications. In terms of the data model, the Sustainable Bridges application has a more specific goal, and, therefore, it can use a specific

TABLE I

DIFFERENCES IN REQUIREMENTS FOR TWO SENSOR NETWORK APPLICATIONS

Property	**Sustainable Bridges**	**Cartalk 2000**
Data Model	Specific	Generic/flexible
Query Model	Push-based	Pull-based
Progr. Paradigm	Publish/Subscribe	Generic query-based
Distr. Transp.	○	●
Energy	●	◔
Mobility	◔	●
Real-time	●	◑
Time Sync	◕	◔
Topology	●	◑

○ Not important ◑ Medium ● Very important

data model, whereas the Cartalk 2000 application needs a generic and flexible data model to support extensibility and generic user interaction. Regarding the query model, for the case of the Sustainable Bridges application, the user only needs to be notified when certain events (material rupture, for example) occur. Therefore, events are pushed to the user, and the application mostly needs to support a publish/subscribe-mechanism. On the other hand, users in the Cartalk 2000 application need to be able to specify their own queries to ask for information such as average speed or road conditions. Therefore, Cartalk 2000 mostly requires a pull-based (query-based) mechanism. In this application only the data and not the node id is important, whereas in the bridge scenario the exact source node of the data has to be known, and so there is no need for distribution transparency in the bridge application. As can be seen from the application descriptions, energy constraints are only important for Sustainable Bridges and mobile nodes only exist within Cartalk 2000. When a node in Sustainable Bridges detects an event, potentially sleeping nodes must be woken up very fast to start their measurements. Thus, real-time constraints are very high. In Cartalk 2000 messages about traffic jams have to be delivered in a reasonable time, too, but not as fast as in Sustainable Bridges. As already mentioned above, the Sustainable Bridges application has very strict time synchronization requirements to ensure good event localization quality, but in the Cartalk 2000 project, less accurate synchronization is sufficient. Regarding topological constraints, the Sustainable Bridges application assumes that sensor nodes are placed manually at critical points of the bridge and so, the exact topology of the network is well-known. In Cartalk 2000, topological information is limited to the use of road and city maps.

C. Requirements for a Generic Framework

To ease the development of sensor network applications, a generic framework is, therefore, necessary. Such a framework has to support the *data-centric model* of sensor network applications and their need for reconfiguration and flexibility. However, sensor networks are heterogeneous and new applications and hardware platforms continuously evolve. Thus, a generic framework has to be *extensible* and *flexible* to manage new application requirements. It should provide mechanisms for the *parametrization of generic components* so that they can meet the requirements of specific applications. If this is not sufficient, new *application-specific components* have to be installed on the sensor nodes. The code of these new components has to be distributed efficiently into the network to avoid wasting energy.

Finally, applications react differently to changes in their environment, e.g., changes in the mobility of nodes. They also have different optimization parameters, e.g., energy or latency. The framework must then be able to *adapt* to these conditions and support optimizations, especially because of the resource limitations found in sensor networks.

III. OVERALL ARCHITECTURE

The overall architecture of `TinyCubus` mirrors the requirements imposed by the applications and the underlying hardware. It has been developed with the goal of creating a generic reconfigurable framework for sensor networks. As shown in figure 1, `TinyCubus` is implemented on top of TinyOS [1] using the nesC programming language [6], which allows for the definition of components that contain functionality and algorithms. We use TinyOS primarily as a hardware abstraction layer. For TinyOS, `TinyCubus` is the only application running in the system. All other applications register their requirements and components with `TinyCubus` and are executed by the framework.

`TinyCubus` itself consists of three parts: the Tiny Data Management Framework, the Tiny Cross-Layer Framework, and the Tiny Configuration Engine, which are described in the following sections.

A. Tiny Data Management Framework

The Tiny Data Management Framework provides a set of standard data management and system components, selected on the basis of the typically data-driven nature of sensor network applications. For each type of standard data management component such as replication/caching, prefetching/hoarding, aggregation, as well

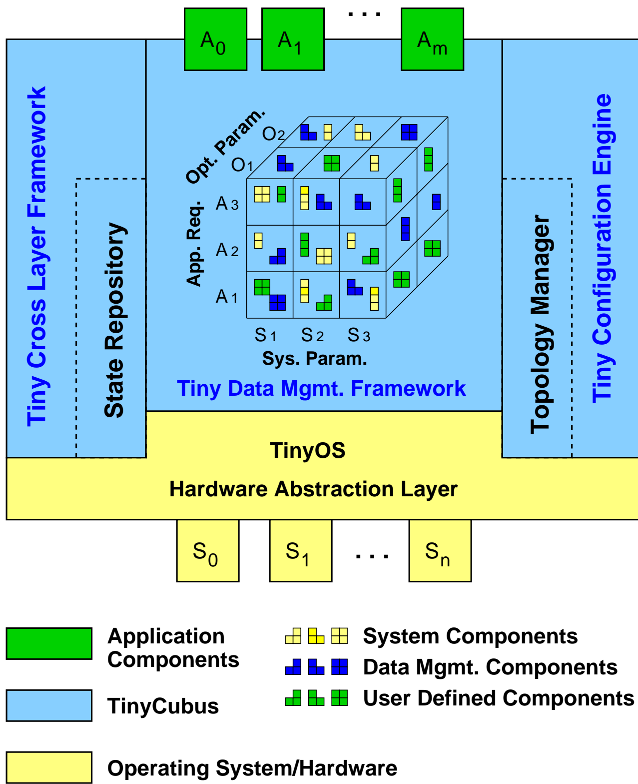

Fig. 1. Architectural components in TinyCubus

as each type of system component, such as time synchronization and broadcast strategies, it is expected that several implementations of each component type exist. The Tiny Data Management Framework is then responsible for the selection of the appropriate implementation based on the information obtained from the system. Of course, only the necessary components are loaded in each sensor and, if other functionality is needed, it can be downloaded from other sensors or gateway nodes connected to larger code repositories.

The cube of figure 1, called 'Cubus', combines *optimization parameters*, such as energy, communication latency and bandwidth; *application requirements*, such as reliability; and *system parameters*, such as mobility. For each component type, algorithms are classified according to these three dimensions. For example, a tree based routing algorithm is energy-efficient, but cannot be used in highly mobile scenarios with high reliability requirements. The component implementing the algorithm is tagged with the combination of parameters and requirements for which the algorithm is most efficient. Eventually, for each combination a component will be available for each type of data management and system components.

The Tiny Data Management Framework selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. This adaptation has to be performed throughout the lifetime of the system and is a crucial part of the optimization process. Therefore, we are currently investigating different strategies that determine when it is necessary – and beneficial – to select a different component. These strategies ensure that the total overhead for adaptation is small compared to the benefits of using the newly selected algorithm.

Furthermore, the parameters and requirements in the three dimensions of the Cubus (system parameters, application requirements, and optimization parameters) have to be carefully selected. Regarding the *system parameters*, we analyze which of them can be measured by a sensor node. In the simplest case these observations are purely local, such as the number of neighbors and their mobility. By examining sensor network applications as outlined in Section II, we determine the *application requirements*. In the broadest sense, they can be subsumed under the term 'quality of service'. Examples are consistency, accuracy, reliability, and real-time constraints. Finally, the *optimization parameters* describe how an algorithm distinguishes itself from other algorithms under the same system and application parameters. These can be latency, communication, and energy. For example, the Sustainable Bridges application requires a time synchronization component for a static scenario that provides high accuracy, but also needs to be optimized with respect to energy use, since sensor nodes are expected to have lifetimes of several years.

B. *Tiny Cross-Layer Framework*

The Tiny Cross-Layer Framework provides a generic interface to support parameterization of components using cross-layer interactions. Strict layering (i.e., each layer only interacts with its immediately neighboring layers) is not practical for wireless sensor networks [7] because it might not be possible to apply certain desirable optimizations. For example, if some of the application components as well as the link layer component need information about the network neighborhood, this information can be gathered by one of the components in the system and provided to all others. Other examples for cross-layer interactions are callbacks to higher-level functions, such as the ones provided by the application developer. The Tiny Cross-Layer Framework provides support for both forms of interaction. It uses a specification language that allows for the description of the

data types and information required and provided by each component. This cross-layer data is stored in the state repository. To deal with callbacks and dynamically loaded code, TinyCubus extends the functionality provided by TinyOS to allow for the dereferencing and resolution of interfaces and components.

1) State Repository: If layers or components interact with each other, there is the danger of loosing desirable architectural properties such as modularity. Therefore, in our architecture the cross-layer framework acts as a mediator between components. Cross-layer data is not directly accessed from other components but stored in the state repository. Thus, if a component is replaced (e.g., to adapt to changed requirements), no component that uses the old component's cross-layer data is affected by the change, given that the new component also provides the same or compatible data. We expect that most components available in the framework will be developed with cross-layer optimizations in mind. Thus, they can (and should) provide cross-layer data even if they do not use it themselves.

Nevertheless, components must know what cross-layer data is available in the state repository. To supply this knowledge we use a specification language which allows to specify what cross-layer data a component needs and provides. With this specification components that make cross-layer data available can also determine if others use their data and if they have to gather it at all.

2) Callbacks: Regarding callbacks to other components, TinyOS already provides some support with its separation of interfaces from implementing components. However, the TinyOS concept for callbacks is not sophisticated enough for our purposes, since the wiring of components is static. With TinyCubus components are selected dynamically and can be exchanged at runtime. Therefore, both the usage of a component and callbacks cannot be static; they have to be directed to the new component if the data management framework selects a different component or the configuration engine installs a replacement for it.

C. Tiny Configuration Engine

In some cases parameterization, as provided by the Tiny Cross-Layer Framework, is not enough. Installing new components, or swapping certain functions is necessary, for example, when new functionality such as a new processing or aggregation function for the sensed data is required by the application. The Tiny Configuration Engine addresses this problem by distributing and installing code in the network. Its goal is to support the configuration of both system and application components with the assistance of the topology manager and role assignment algorithms.

1) Topology Manager: The topology manager is responsible for the self-configuration of the network and the assignment of specific roles to each node. A role defines the function of a node based on properties such as hardware capabilities, network neighborhood, location etc. Examples for roles are SOURCE, AGGREGATOR, and SINK for aggregation, CLUSTERHEAD, GATEWAY, and SLAVE for clustering applications as well as VIBRATION to describe the sensing capabilities of a node. In previous work [8] we describe a generic specification language and an algorithm for efficient role assignment that are briefly outlined in the remainder of this section.

Since in most cases the network is heterogeneous, the assignment of roles to nodes is extremely important: only those nodes that actually need a component have to receive and install it. As we show in Section IV, this information can be used by the configuration engine, for example, to distribute code efficiently in the network.

2) Role Specification and Role Assignment Algorithm: For role assignment the topology manager uses a generic specification language and a decentralized role assignment algorithm. In the specification language a role is defined by a rule. If a rule is satisfied, the algorithm assigns the role to the node. For example, the following rule assigns the role CLUSTERHEAD if there is no other node with this role in the 1-hop neighborhood:

```
CLUSTERHEAD :: {
  count(1-hop) {role == CLUSTERHEAD} == 0
}
```

Copies of the role specification have to be present on all nodes because the role assignment algorithm is executed on each of them. Whenever possible it only uses local knowledge. However, if information about the network neighbors is required (e.g., the number of nodes in the neighborhood with a given role), the node has to retrieve this information from its neighbors while avoiding conflicting role assignments (see [8] for details).

IV. Role-Based Code Distribution Algorithm

In many sensor network applications the topology of the roles in the network is known in advance and follows a regular structure. This is definitely the case if roles are defined with routing in mind, such as with clustering approaches. Of course, in the general case, roles can be based on other properties of the application or the system

at hand. A good example is provided by the Sustainable Bridges application (Fig. 2), where nodes affixed to the edge are equipped with vibration sensors, whereas others are only required to provide temperature readings.

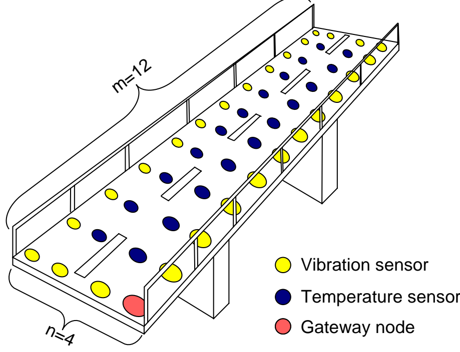

Fig. 2. Sensor topology for Sustainable Bridges

Having information about different roles, and assuming that, in most cases, a difference in role assignment is motivated by differences in functionality, a code distribution algorithm can leverage this knowledge to route code updates only through the set of nodes that really need it, that is, belong to a specific role. In other words, if the code for nodes with vibration sensors is updated, for example, because a new in-network vibration data processing algorithm is needed, this should not affect the temperature nodes available in the system. Of course, the code distribution algorithm has to make sure that all nodes receive the appropriate messages reliably so that, in the end, they all run the same version of the application.

Thinking of the severe energy constraints of sensor nodes in this particular application, and taking into account that the energy cost for data transmission is very high, a scheme that can reduce the number of messages sent unnecessarily to irrelevant nodes is beneficial.

A. Network Model

Let us now discuss the network model used by the distribution algorithm, before we get to its detailed description in the next section. For our algorithm, a network consists of a set of inner nodes $I = \{n_0, \cdots, n_i\}$ and a set of gateway nodes $G = \{g_0, \cdots, g_j\}$ through which messages from outside the sensor network, such as code updates, are inserted into the system. Let $N = I \cup G$ be the set of all nodes in the network so that $I \cap G = \emptyset$. The set of roles R is defined as $R = \{r_0, \cdots, r_m\}$, and $A : N \rightarrow R$ defines a complete relation that assigns roles to nodes. For all $r \in R$ let $N_r = \{n \in N : A(n) = r\}$ be the set of nodes assigned to role r.

We further define the *1-hop neighborhood* of a node n_i as the set of nodes in the single-hop broadcast range of n_i, as follows:

$$H_1(n_i) = \{n_j : n_j \text{ is in broadcast range of } n_i\}$$

The *at most k-hop neighborhood* is defined recursively using the expression:

$$H_k(n_i) = \bigcup_{n_j \in H_{k-1}(n_i)} H_1(n_j),\ k \geq 2$$

The set of nodes with role r in at most k-hop distance from node n_i is simply:

$$H_{r,k}(n_i) = H_k(n_i) \cap N_r$$

We say that a node n_i is *at most k-hop connected* with another node n_j with role r iff n_i and n_j are connected through nodes $n_{l_1} \dots n_{l_m}$ with role r, so that $n_{l_1} \dots n_{l_m} \in N_r$ and the path between two consecutive nodes in this set, between n_i and n_{l_1}, and between n_{l_m} and n_j involves at most $k-1$ nodes $\notin N_r$. The following equations define the transitive closure of all reachable nodes *at most k-hop connected* with n_i for a particular role r. Note that if n_i is not of role r it is not included in $C^p_{r,k}(n_i),\ p \geq 1$.

$$\begin{aligned} C^0_{r,k}(n_i) &= \{n_i\} \\ C^p_{r,k}(n_i) &= \bigcup_{n_j \in C^{p-1}_{r,k}(n_i)} H_{r,k}(n_j),\ p \geq 1 \\ C_{r,k}(n_i) &= C^p_{r,k}(n_i),\ \text{iff } C^p_{r,k}(n_i) = C^{p+1}_{r,k}(n_i) \end{aligned}$$

Assuming that messages are inserted at all gateway nodes, the following equation defines the set of nodes with role r that can be reached with *at most k-hop connectivity*:

$$NC_{r,k} = \bigcup_{g_i \in G} C_{r,k}(g_i)$$

For every role, the smallest k is calculated so that all nodes in the network of this role are *at most k-hop connected*: $k_r = \min\{k : NC_{r,k} = N_r\}$.

The value k_N for which all nodes of all roles in the network are connected is then calculated as: $k_N = \max\{k_{r_1}, \cdots, k_{r_m}\}$

B. Detailed Description

Our code distribution algorithm uses this network model and the information about role assignments provided by the Tiny Cross-Layer Framework to efficiently disseminate code updates to specific roles. The algorithm

starts at gateway nodes by broadcasting data to its k_r-hop neighborhood. Then, only nodes with role r forward this data further to their own k_r-hop neighbors, thus flooding the nodes with role r while using only those nodes with other roles that are necessary to reach them. The algorithm can be parametrized by selecting k_r for each role. The topology of the network is, therefore, crucial. If, such as for the case depicted in Fig. 2, the network is at most k_r-hop connected for a given role r, where $k_r = 1$, it is possible to reach all target nodes with maximum efficiency. However, in the general case, especially if topologies are random or $k_r > 1$, other nodes with roles different from r need to be involved in the process of forwarding this information.

This is only true, of course, if it is necessary to guarantee delivery to all nodes with a given role. For cases where 100% delivery is not required, a smaller $k : k < k_r$ can be selected to provide more energy-efficient processing, at the cost of sacrificing completeness, as we will see in more detail in the next sections. For this reason, our distribution algorithm is parametrized with respect to k and allows the application, or other components in the system, to select the level of completeness by choosing the appropriate k.

In addition, if reliability is necessary, such as for the case of providing code updates, the distribution algorithm makes use of implicit acknowledgments. If a neighbor forwards a message sent by node n_i, n_i treats this message as an acknowledgment. If after a certain amount of time, the neighbor does not forward the message, n_i retransmits it. Following the modularization techniques advocated at the beginning of this paper, this reliability component of our algorithm can be replaced with any other scheme that ensures reliable transmissions.

Finally, in our algorithm, a node n_i waits a random time $t \in [0, \ldots, t_{max}]$ before retransmitting a message. This is just one possible way to avoid the broadcast storm problem, mentioned in [9] and, like the reliability component, can be replaced with any other scheme that avoids collisions. Of course, the choice of t_{max} is directly related with the delay observed in the evaluation of the algorithm.

In summary, our role-based dissemination algorithm has four settable parameters that, in our system, are maintained by the Tiny Cross-Layer Framework: r, the role of the target nodes; k_r, the network connectivity used for broadcasting data; the boolean variable rel that controls whether or not implicit acknowledgments will be used; and t_{max} that determines the maximum retransmission time.

Assumptions: In the implementation of our algorithm, we assume that roles have already been assigned and that there is no dynamic reassignment of roles while the code dissemination algorithm runs. This means that the connectivity k_r of the network for a given role r can be determined up-front. Furthermore, we assume that nodes are stationary, do not fail, and have already determined their neighborhood $H_{r,k}(n)$ with respect to a given role r and network connectivity k. Finally, communication is assumed to be performed via bidirectional local broadcasts and that transmission failures, if they occur, are not permanent.

C. Evaluation

In order to show the feasibility of our approach, we have implemented the role-based code distribution algorithm for motes running TinyOS [1]. In the first set of experiments, we show analytically and by means of experiments the worst case and average results for the computation of k_r, the connectivity of a role, both for structured and random grid-like topologies of sensor networks. In the second set of experiments, we compare the efficiency of our algorithm with a flooding approach that has been modified to provide reliability and collision avoidance. The results presented in this paper have been obtained using TOSSIM, the TinyOS simulator provided by UC Berkeley [10].

1) Experimental Setup: In our experiments, we have analyzed two scenarios. In the first one, the sensor nodes are laid out in an evenly spaced $m \times n = 12 \times 4$ grid with the role assignment depicted in Fig. 2, which represents the topology of the Sustainable Bridges application. There is only one gateway node g_0, located in one of the corners, used to inject messages to the network. The distance between the nodes is 10 meters and their radio model is set to a lossless disc model with a communication range of 15 meters. Finally, packet losses occur only due to collisions and the maximum retransmission delay t_{max} has been set to 150 ms and 600 ms respectively. In the second scenario, the same parameters apply, except that roles, instead of being sorted into a regular structure, are randomly assigned.

In our scenarios, we assume the presence of two roles: `VIBRATION` and `TEMPERATURE`, that represent the two types of sensors found in the network. We evaluate the code distribution algorithm by sending (fictitious) code updates from the gateway node g_0 to all vibration sensors. For the purpose of this paper, we are mostly interested in the efficiency of the algorithm in terms of

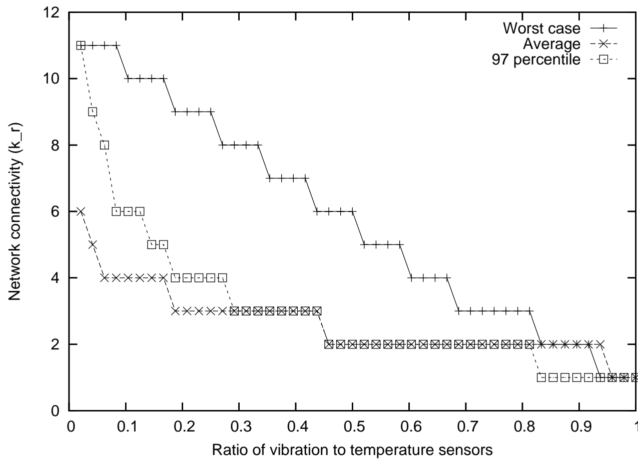

Fig. 3. k_r needed to achieve 100% completeness (random scenario)

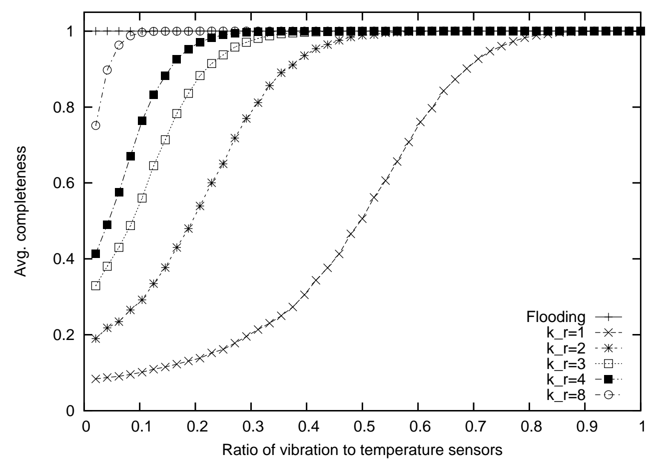

Fig. 4. Avg. completeness for k_r (random scenario)

the number of forwarded messages, and do not aim yet at measuring the performance of real code updates.

2) Computation of k_r: The first set of experiments deals with the computation of a reasonable k_r, that is, the connectivity of the network for a given role, as seen from the perspective of gateway nodes. In the case of applications such as the Sustainable Bridges project, the computation of k_r can be performed by hand by the application developer. The structure of the network as well as the location of the sensor nodes is well known. In our case, k_r is known to be 1 for the vibration nodes.

However, if the structure of the network is random or not known a priori, a way of determining "good" values of k_r is desirable. Fig. 3 shows the worst case, average and 97 percentile values of k_r for a network with random role assignments, if we choose k_r so that every node is reached. From the graph we see that, starting from a ratio of vibration to temperature nodes of 30%, $k_r = 3$ achieves 100% completeness in both the average and 97 percentile cases, even though the worst case indicates a value of at least $k_r = 8$. The curves of Fig. 3 have been obtained by choosing 10000 random role assignments in a $m \times n = 12 \times 4$ grid, varying the number of `VIBRATION` nodes from 1 to 48 and measuring the value of k_r needed to achieve 100% completeness. For the worst case curve, it is possible to compute a general expression that gives the values of k_r for arbitrary grid structures of size $m \times n$. Assuming w.l.o.g. that $n \leq m$ and that nodes can communicate with their immediate horizontal, vertical and diagonal neighbors and that the only gateway node g_0 is located in one of the corners, the analytical formula for the worst case is:

$$k_r = \begin{cases} m - \lceil \frac{|N_r|}{n} \rceil & \text{if } 1 \leq |N_r| \leq (m-n)n \\ \lfloor \sqrt{(mn - |N_r|)} \rfloor & \text{if } (m-n)n < |N_r| < mn \\ 1 & \text{if } |N_r| = mn \end{cases}$$

There might, however, be situations where 100% completeness is not required. For these cases, Fig. 4 shows the average completeness achieved for $k_r = 1, 2, 3, 4$ and 8. Using this graph we can determine the smallest value of k_r needed to achieve the desired completeness, assuming that a given ratio of nodes with the target role is known. For example, if we have a ratio of target roles equal to 10% and would like to achieve at least 80% completeness, Fig. 4 tells us that with $k_r = 4$, we can achieve on average the desired results.

3) Performance Results: In the second set of experiments, we have focused on evaluating the performance of our role-based distribution algorithm using both scenarios described above.

Fig. 5 shows the number of messages sent on average by each node in the Sustainable Bridges scenario. The graph compares the messages sent by both flooding and our role-based distribution algorithm for maximum retransmission delay $t_{max} = 150ms$ and $600ms$, respectively. Role assignments on the x-axis vary from the original configuration depicted in Fig. 2 to all nodes being assigned the `VIBRATION` role. The measurements shown are the average of 100 runs. In the graph, we can see that flooding with $t_{max} = 150ms$ requires about 5 messages per node, whereas with $t_{max} = 600ms$, it requires only a little over 2 on average. Since the flooding algorithm retransmits messages in the presence of collisions until all nodes are reached, the average

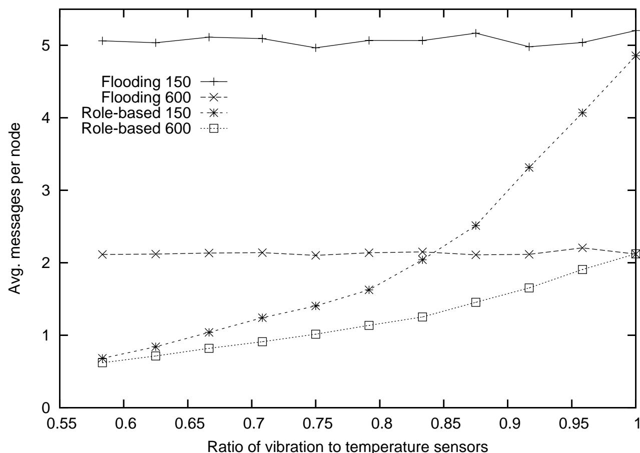

Fig. 5. Avg. number of sent messages per node (structured scenario)

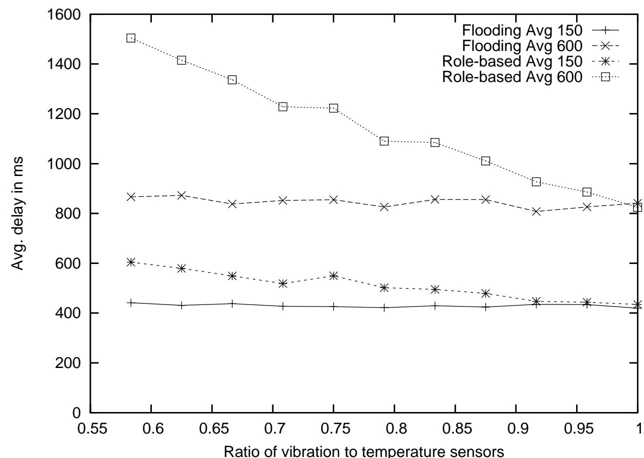

Fig. 6. Avg. delay for message delivery (structured scenario)

number of messages sent is greater than 1 and varies with the length of t_{max}. In addition, the graph shows that the number of messages sent is independent of the ratio of vibration to temperature sensors, since flooding does not distinguish among them to distribute data.

In contrast, our role-based algorithm performs much better than flooding, especially when the ratio of vibration to temperature sensors is low, since only vibration sensors are required to forward messages[1]. As expected, the number of messages per node increases as the ratio of vibration to temperature sensors increases. In the extreme (when all nodes in the network are vibration nodes), our algorithm behaves just like flooding.

Fig. 6 depicts the average delays needed by both algorithms to reach all vibration nodes in the structured scenario. Maximum delays (not shown in the graph) are for our algorithm in the worst case as much as twice as long as the delay needed on average. In addition, average delays for flooding are at most 1.5 times better than our role-based algorithm. The reason is that flooding uses not only vibration nodes to forward the data (which allows for more parallelism), and the fact that in our network all vibration nodes are located in a square so that, if one vibration node chooses a long random delay to avoid collisions, data distribution as a whole is delayed. Nevertheless, by choosing for example $t_{max} = 150ms$, it is possible to keep the number of sent messages low (see Fig. 5), while achieving delays just slightly above those of flooding (compare Flooding Avg 150 and Role-based Avg 150 in Fig. 6).

As we have just seen, our role-based distribution algorithm can be used very efficiently with structured topologies, such as the one of Sustainable Bridges, but one cannot always expect to have topologies that exhibit 1-hop connectivity. Therefore, we have tested our algorithm with random distributions of roles to show that it can also be used effectively in such scenarios. Fig. 7 shows the number of messages sent for random role assignments by both, flooding and two different versions of our role-based algorithm with $k_r = 1$ and $k_r = 2$ respectively. Flooding behaves, as expected, just like in the structured case. Our algorithm, on the other hand, sends far fewer messages than flooding, but if the topology of the network exhibits, say 4-hop connectivity for role r, our algorithm will not reach all nodes.

As shown in Fig. 4, flooding obviously always reaches 100% completeness, but our algorithm cannot guarantee completeness in all cases. If, for example, a 1-hop algorithm is used and the network exhibits 3-hop connectivity, not all required nodes will be reached. The use of a 2-hop algorithm, however, reaches 100% completeness with very high probability, if the ratio of vibration sensors is greater than 55% while requiring a relatively low number of messages (see Fig. 7). For example, a ratio of 55% vibration nodes sends on average 1.6 messages per node, whereas flooding requires 2.1. Therefore, even in cases where the connectivity of the network is not known, we can use Fig. 4 to choose a reasonable value of k_r based on the desired level of completeness.

Finally, analogously to the structured case, Fig. 8 shows the average delays needed by flooding, the 1-hop and 2-hop algorithms to reach the target nodes. For the 400 different random role assignments tested,

[1]Recall that the network topology in this scenario exhibits 1-hop connectivity for the VIBRATION role.

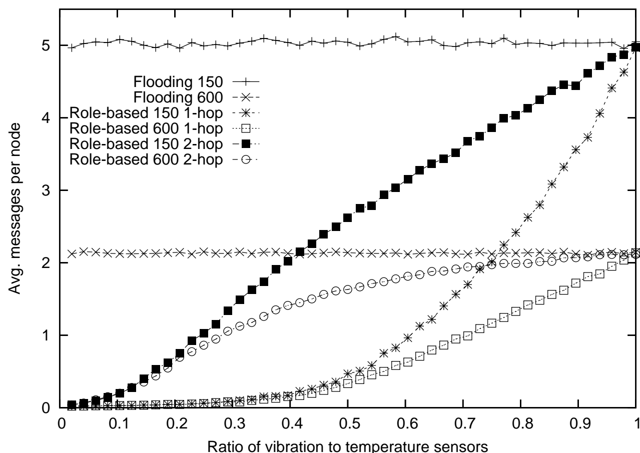

Fig. 7. Avg. number of sent messages per node (random scenario)

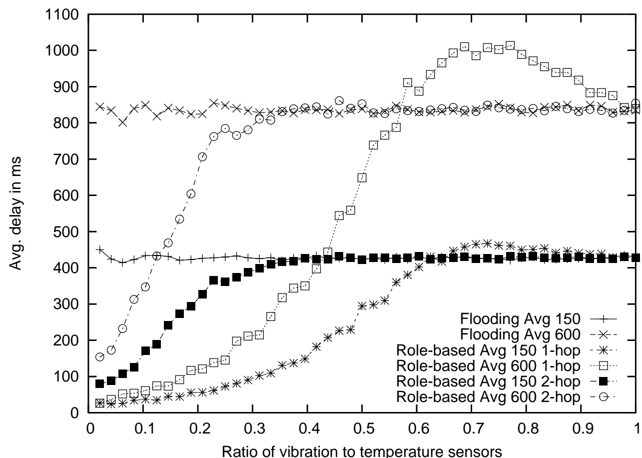

Fig. 8. Avg. delay for message delivery (random scenario)

our algorithm has lower delays than flooding for low ratios of vibration sensors but, as shown in Fig. 4, at those values our algorithm on average does not reach all target nodes, so it has a clear (and unfair) advantage. As soon as the number of vibration sensors reaches a point of saturation where our algorithm can probabilistically reach all nodes (ratio of 0.55 for the 2-hop algorithm), it behaves similarly to flooding. Between ratio values of 0.6 and 1, our 1-hop algorithm is at most 20% slower than flooding, which correlates with the results of Fig. 6. In this case, however, the differences in delay are not as noticeable. The reason for this is that, when using random assignments, roles are placed arbitrarily and therefore, the number of neighboring nodes with the same role is, on average, higher than for the bridge scenario, where each vibration sensor only has two direct neighbors. Therefore, the delays presented in Fig. 6 represent the worst case scenario, where each message reaches exactly one target and thus, data is sent serially from one vibration sensor to the next.

D. Advantages of Role-Based Code Distribution

As we have seen in the evaluation section, the results provided by our role-based algorithm are very promising for structured scenarios. For these cases, we can use application knowledge about the topology of the network to improve on the number of messages sent while maintaining reliability. We have also shown that it is relatively easy to determine probabilistically reasonable values of k_r even for networks where the topology is not known or exhibits random properties so that, even in these cases our algorithm outperforms reliable plain flooding techniques. However, many sensor networks are deployed in environments where either the structure or the topology of the network is well-known and so, this knowledge can be trickled down to the routing components to provide more efficient data distribution. Therefore, using the terminology introduced at the beginning of this paper, the experimental results described above could allow the Tiny Data Management Framework to classify our role-based algorithm within the Cubus (see Fig. 1) so that it can be selected when the appropriate system parameters, application requirements and optimization criteria require it.

Additionally, our role-based code distribution algorithm has several advantages. In general, the algorithm can be used to distribute any kind of data whose destination varies based on certain information, such as roles. Furthermore, if we assume that roles have already been assigned and that the role connectivity k_r of the network has been determined (or estimated), our algorithm is more efficient than plain flooding. Moreover, if we assume that the network topology does not change too much or is even static, a one-time overhead for the computation of k_r is a small penalty to pay for continuously sending data with several times less overhead than reliable flooding.

Even in the case where nodes are mobile and their distribution changes, if we assume that the ratio of r to all other nodes is high enough, we can use the results detailed above to estimate values of k_r that work well. Furthermore, each node might decide to perform this estimation and conclude that its own neighborhood is relatively static with respect to changes in the topology, and that certain values of k_r work even in the presence of mobility.

The fact that the algorithm is parametrized with respect to the properties of the network allows us to select the appropriate version depending on the desired result. There is, therefore, a tradeoff between latency and the number of messages that can be used by our framework to adapt to the requirements of the application or the network itself.

Finally, although the experiments presented in this paper only deal with two distinct roles, our results are valid for any number of roles. The only difference is in the definition of the ratio, which is generally determined by the number of nodes of a given role r divided by the sum of all nodes with roles different from r.

V. Related Work

`TinyCubus` and our role-based code distribution algorithm are related to a variety of other work. In this section, we provide a description of relevant projects that are in the process of creating frameworks similar (in part) to ours, related code distribution schemes, and finally, routing algorithms that, like ours, use cross-layer data to make forwarding decisions.

SensorWare [11] and Impala [12] aim at providing functionality to distribute new applications in sensor networks, just like our configuration engine. For this purpose, they create abstractions between the operating system and the application, although both differ slightly from each other. SensorWare uses a scripting language that is not really well-suited for resource-limited platforms such as our TinyOS motes. It uses special commands of the language that allow the forwarding of the current program to other nodes, and tries to avoid unnecessary code transfers by transmitting the code only if the script is not already running on the neighboring nodes. However, SensorWare does not support adaptation and cross-layer interactions, as proposed in our framework.

In Impala, new code is only transmitted on demand if there is a new version available on a neighboring node. Furthermore, if certain parameters change and an adaptation rule is satisfied, the system can switch to another protocol. However, this adaptation mechanism only supports simple adaptation rules. Although it uses cross-layer data, Impala does not have a generic, structured mechanism to share it and so, is not easily extensible.

The MobileMan project [13] is a system that aims at creating a cross-layer framework similar to ours. However, MobileMan is not targeted towards sensor networks and assumes environments typical of mobile ad-hoc networks, which are, in the general case, not so limited in terms of resources. In addition, MobileMan focuses on data sharing between layers of the network protocol stack and, therefore, does not include the configuration and adaptation capabilities found in our framework.

Finally, EmStar [14] is a software environment for Linux-based sensor nodes that, like MobileMan, assumes the presence of higher-end nodes as part of the sensor network. Similar to our data management framework, EmStar contains some standard components for routing, time synchronization, etc. but is not able to provide the adaptation mechanisms available in our framework.

Regarding related work concerned with the implementation of code distribution, Ripple [15] is a code distribution algorithm implemented using EmStar. In order to reduce the number of messages sent, this algorithm uses a publish/subscribe scheme where a single node in the neighborhood sends code updates to its subscribers. Similar to our approach, it includes a mechanism to transmit code updates reliably, but it fails to consider cross-layer data (e.g., role information) and, therefore, data is always forwarded to all nodes.

Another example of code propagation for sensor networks is Trickle [16]. Trickle periodically broadcasts meta-data about the software version nodes are using, and focuses on detecting whether or not a code update is needed. On the other hand, our role-based algorithm is used to selectively send code updates to nodes that are supposed to receive it based on their role assignmnent. Of course, it would be possible to combine both algorithms to further optimize code updates in our system.

Reijers and Langendoen [17] describe a scheme to install code on sensor nodes. Their goal is to minimize the size of the code image by transmitting only the differences to the previous version, which is something not considered in our scheme. However, they do not address how updates are distributed in the network.

Finally, there are a number of routing algorithms [18], [19] that use cross-layer information to improve on their efficiency, although this is usually done on a protocol-specific basis. One example is the use of spatial information for routing, as has been done in the Cartalk 2000 project [20]. However, Cartalk does not provide a generic mechanism to allow for arbitrary cross-layer data sharing that can be used with other schemes.

VI. Conclusion and Future Work

In this paper, we have described the architecture of `TinyCubus`, a flexible, adaptive cross-layer framework for sensor networks. Its specific requirements have been derived from the increasing complexity of the hardware capabilities of sensor networks, the variety and breadth

found in typical applications, and the heterogeneity of the network itself. Therefore, we have designed our system to have the Tiny Data Management Framework, that provides adaptation capabilities, the Tiny Cross-Layer Framework, that provides a generic interface and a repository for the exchange and management of cross-layer information, and the Tiny Configuration Engine, whose purpose is to manage the upload of code onto the appropriate sensor nodes.

Furthermore, we have provided the description of a novel role-based code distribution algorithm that uses cross-layer information, such as role assignments, in order to improve on the number of messages needed to distribute code to specific nodes. The results of our evaluation show that this algorithm performs several times better than plain flooding in scenarios where the topology and distribution of roles within the network is well-known. For situations where this is not the case, we have provided analytical results that allow us to compute the connectivity of the network for a given role with minimal overhead, and have shown that our algorithm can be adapted to different network topologies and still provide an improvement over flooding schemes.

The implementation of `TinyCubus` is still under way and, although the prototypes for the cross-layer framework and configuration engine are already partially functional, there is still work to do. We are in the process of integrating our framework with an additional application that provides the capabilities found in a smart environment and that will fully make use of the functionality provided by `TinyCubus`.

Finally, regarding the role-based code distribution algorithm, we plan on extending it to support highly mobile sensor nodes, like the ones found in the Cartalk 2000 project, and to include functionality found in related projects, like Trickle. In addition, we would like to analyze other types of topologies where nodes are not equally spaced and determine how well our role-based algorithm works under such conditions.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” in *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[2] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, “Habitat monitoring with sensor networks,” *Comm. of the ACM*, vol. 47, no. 6, pp. 34–40, 2004.

[3] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, K. Rothermel, and C. Becker, “Adaptation and cross-layer issues in sensor networks,” in *Proc. of the Intl. Conf. on Intelligent Sensors, Sensor Networks & Information Processing*, 2004.

[4] Sustainable bridges web site. [Online]. Available: http://www.sustainablebridges.net

[5] D. Reichardt, M. Miglietta, L. Moretti, P. Morsink, and W. Schulz, “CarTALK 2000: Safe and comfortable driving based upon inter-vehicle-communication,” in *Proc. of the Intelligent Vehicle Symp.*, vol. 2, 2002, pp. 545–550.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” in *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation*, 2003, pp. 1–11.

[7] A. J. Goldsmith and S. B. Wicker, “Design challenges for energy-constrained ad hoc wireless networks,” *IEEE Wireless Communications*, vol. 9, no. 4, pp. 8–27, 2002.

[8] K. Römer, C. Frank, P. J. Marrón, and C. Becker, “Generic role assignment for wireless sensor networks,” in *ACM SIGOPS European Workshop*, 2004, to appear.

[9] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu, “The broadcast storm problem in a mobile ad hoc network,” *Wireless Networks*, vol. 8, no. 2/3, pp. 153–167, 2002.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler, “TOSSIM: Accurate and scalable simulation of entire TinyOS applications,” in *Proc. of the 1st Intl. Conf. on Embedded Networked Sensor Systems*, 2003, pp. 126–137.

[11] A. Boulis, C.-C. Han, and M. B. Srivastava, “Design and implementation of a framework for efficient and programmable sensor networks,” in *Proc. of the 1st Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys 2003)*, 2003.

[12] T. Liu and M. Martonosi, “Impala: A middleware system for managing autonomic, parallel sensor systems,” in *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003, pp. 107–118.

[13] M. Conti, G. Maselli, G. Turi, and S. Giodano, “Cross-layering in mobile ad hoc network design,” *IEEE Computer*, vol. 37, no. 2, pp. 48–51, 2004.

[14] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, “EmStar: A software environment for developing and deploying wireless sensor networks,” in *Proc. of USENIX 2004*, 2004, pp. 283–296.

[15] T. Stathopoulos, J. Heidemann, and D. Estrin, “A remote code update mechanism for wireless sensor networks,” University of California, L.A., Tech. Rep. CENS-TR-30, November 2003.

[16] P. Levis, N. Patel, D. Culler, and S. Shenker, “Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks,” in *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*, 2004.

[17] N. Reijers and K. Langendoen, “Efficient code distribution in wireless sensor networks,” in *Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Appl.*, 2003, pp. 60–67.

[18] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, “Energy-efficient communication protocol for wireless microsensor networks,” in *Proc. of the Hawaii Intl. Conf. on System Sciences*, vol. 2, 2000, p. 10 ff.

[19] W. H. Yuen, H. no Lee, and T. D. Andersen, “A simple and effective cross layer networking system for mobile ad hoc networks,” in *Proc. of the 13th IEEE Intl. Symp. on Personal, Indoor and Mobile Radio Communications*, vol. 4, 2002, pp. 1952–1956.

[20] J. Tian, L. Han, K. Rothermel, and C. Cseh, “Spatially aware packet routing for mobile ad hoc inter-vehicle radio networks,” in *Proc. of the IEEE 6th Intl. Conf. on Intelligent Transportation Systems (ITSC)*, vol. 2, 2003, pp. 1546–1551.