Management and Configuration Issues for Sensor Networks

Pedro José Marrón[*], Andreas Lachenmann, Daniel Minder,
Matthias Gauger, Olga Saukh, and Kurt Rothermel

Institute of Parallel and Distributed Systems (IPVS)
University of Stuttgart, Germany
{marron, lachenmann, minder, gauger, saukh, rothermel}@informatik.uni-stuttgart.de

Short abstract

We define three of the key issues related to efficient management and configuration of sensor networks: the distribution and management of roles within the network, efficient code distribution algorithms, and efficient on-the-fly code update algorithms. We present some results for each of these issues as we have dealt with them within the `TinyCubus` project.

Full abstract

In this paper, we define three of the key issues that need to be solved in order to provide efficient management and configuration of applications and system software in sensor networks: the distribution and management of roles within the network, efficient code distribution algorithms, and efficient on-the-fly code update algorithms for sensor networks. The first issue is motivated by the increasing heterogeneity of sensor network applications and their need for more complex (non-homogeneous) network topologies and structures. The second one is motivated by the intrinsic energy constraint issues and, in general, the resource limitation of sensor networks. Finally, the third one is needed due to the nature of monitoring applications and optimization needs from applications that should be able to efficiently incorporate code updates so that the network can adapt to its surroundings on the fly. In this paper we present related work and some results for each of these issues as we have dealt with them within the `TinyCubus` project.

[*]Corresponding author, marron@informatik.uni-stuttgart.de, IPVS, University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany, Phone +49-711-7816-223, Fax +49-711-7816-424

1 Introduction

Sensor networks are envisioned to consist of a large number – possibly thousands – of sensor nodes that communicate with each other using wireless technology. From a network management perspective, it is important that these nodes can manage and configure themselves autonomously. Otherwise, configuration management for such numbers of nodes cannot be handled effectively. Even today, where most sensor networks consist of only tens of nodes, applications are becoming more and more complex, so that configuration and reconfiguration during normal operation play a very important role.

The problem of configuring sensor networks can be formulated as follows. The software of individual sensors has to be set up efficiently so that the network can fulfill its tasks, given the resource constraints typically present in sensor networks. If this configuration process is repeated after the initial deployment of the nodes, we call it "reconfiguration".

One example for configuring sensor networks is the assignment of specific tasks to individual sensor nodes. This configuration can be done either manually or – if this is not feasible as it is the case in large networks – by having the network itself assign these tasks. Another example is the distribution of code updates in the network. With potentially thousands of nodes, it would be impracticable if a human network administrator had to install a new software version on each node by hand. Therefore, some system has to be implemented that supports and automates this process in order to distribute code updates in the network using its multi-hop capabilities. In addition, since energy is an extremely scarce resource, new software versions have to be disseminated in an energy-efficient manner, i.e., with the transmission of as few packets as possible. Furthermore, if code is sent over a radio link, the complete replacement of code images of complex applications with every software update is inefficient because many packets are used to transfer code that was also part of the previous version and, therefore, is already present in the network.

In this paper, we describe three of the most relevant research problems that need to be solved in order to support autonomous configuration and efficient on-the-fly reconfiguration of sensor networks and show some results for each of them. The first challenge deals with the assignment of roles to sensor nodes based on the properties of the network. These roles represent the task that a specific node is supposed to fulfill. For example, a node with an `AGGREGATOR` role aggregates the data received from its neighbors into a single result whereas a node with the role `DATASOURCE` just collects data from its sensors and forwards them to an aggregator node. The second challenge deals with the update of code running on the nodes in an energy-efficient way. Most current approaches send the same code image to the whole network. However, leveraging knowledge about the topology and the specific role assignment available in the network, the number of messages sent to a node can be significantly reduced by sending code updates to only those nodes that really need them. Finally, the third challenge deals with the minimization of the size of such code updates and providing the flexibility needed to dynamically adapt the software running on the nodes. Always replacing complete code images – even for just small changes – requires lots of unnecessary transmissions. If only those parts of the code that have actually changed are transmitted, the number of sent packets can be reduced significantly. In addition, if several pieces of code providing the same functionality with different implementations are present on a node, the node can dynamically adapt the application to the current requirements.

These issues in combination with dynamic adaptation issues are the main focus of the `TinyCubus` project at the University of Stuttgart.[11, 12] `TinyCubus` is a flexible, adaptive cross-layer framework for sensor networks, whose goal is to ease the development of a broad range of sensor network applications by providing the necessary infrastructure to deal with the complexity of such systems.

The remainder of this paper is organized as follows. The next section describes the three main challenges in configuration management and gives an overview of related work for each of them. Section 3 presents the `TinyCubus` architecture and our approach to solving these problems. Finally, section 4 concludes this paper.

2 Challenge Description and Related Work

As already mentioned in the introduction, we have identified three of the main challenges that are involved in the configuration of sensor networks: the assignment of roles within the network, efficient code distribution techniques, and on-the-fly code update algorithms. Let us now discuss each problem in more detail.

2.1 Role Assignment

Most current sensor network research assumes that all nodes are equal. However, in practice, sensors often have to fulfill different tasks. For example, a sensor node might act as a cluster head whereas its neighbors could act as slaves or gateways to other clusters. Another example often found in sensor networks is in-network aggregation. Most nodes just collect data from their sensors and transmit it to aggregator nodes. These nodes aggregate the data of several sources in order to reduce the number of messages required or to increase the accuracy of the values. Finally, a sink node consumes the aggregated data.

As these examples show, the tasks of individual nodes in the network differ considerably. Therefore, the assignment of roles that represent these tasks (e.g., `GATEWAY`, `DATASOURCE`, `SINK`) needs to be addressed. There are several approaches to solve this problem in current sensor networks.

One solution is to have a (human) network administrator manually assign a role to each node. Clearly, this approach only works if the network is small and if it is possible to obtain global knowledge about the topology of the network. For example, in the Sustainable Bridges application,[12, 17] which uses sensor networks to cost-effectively monitor bridges in order to detect structural defects, the network is small, nodes are static, and the engineers want to have a strict control over the whole network. Therefore, in this case such an approach is viable. However, if nodes move or fail, the system cannot adapt to the new situation without human intervention, which is obviously a disadvantage.

Another widely spread solution is to use the differences in software functionality to assign different roles. Here, the hardware of all nodes is the same but different software is loaded onto them so that they implicitly have different roles. For example, in TinyOS some of the demo applications rely on a special program, TOSBase, being loaded on one of the motes, that just forwards all messages received over the radio to the PC via a serial cable. The actual application that reads sensor values and sends them over the radio runs only on the other nodes.

Somehow related is an approach that implicitly differentiates roles based on the hardware capabilities of the nodes. If the hardware properties differ, this distinction can be used to identify the roles. For example, in the Sustainable Bridges project, not all nodes have the same sensing capabilities. Vibration sensors are affixed near the edges whereas other nodes in the center of the bridge just monitor the environmental conditions using temperature sensors.[12] With this approach role distribution is very easy but adaptation is not because roles are not interchangeable.

In the last few years several algorithms for self-configuration of sensor networks have been proposed. These solutions make sure that the nodes divide the tasks among themselves. One example is the distributed placement of aggregation operators.[1] The placement of the operators is incrementally adjusted to minimize network traffic. Obviously, this solution only focuses on a specific part of the self-organization problem rather than providing a generic solution.

Similarly, Kochhal et al.[8] describe a protocol for self-organization that uses roles to describe the tasks of specific nodes. However, their approach does not provide a generic solution, either, because they only use a fixed set of roles.

None of these approaches truly supports self-configuration in a generic way. Such a solution would be able to assign a role representing any task based on the capabilities of a sensor node, its neighbors, and the topology of the network. We propose a specification language and a distributed role assignment algorithm that can be used to assign arbitrary roles to nodes based on these properties. When the properties change, the role assignment is dynamically updated to reflect the new situation.

2.2 Code Distribution

Until recently, if a new software version was to be installed in a sensor network, every node had to be plugged into a programming board connected to a PC using a serial or parallel cable. Since then, several code distribution techniques have been proposed that transmit code images from node to node using radio links. These schemes are extremely useful for the installation of new software in a large network after having deployed it in the field. Typically, a code image for Mica2 motes running TinyOS[4] consists of several kilobytes of data. Having in mind that radio communication needs much of the limited energy resources, the main challenge is to minimize the number of packets sent to each node while delivering code updates reliably to all nodes that need them.

A simple solution would be to use a flooding technique that blindly sends the code update to all nodes. However, with such an approach it is not possible to install code updates selectively on only some nodes in the network in an efficient way. Instead, all nodes have to forward all packets, even if they have been assigned a role that does not require the specific code update to fulfill its task. Therefore, in the presence of roles, such an approach is not particularly efficient.

Several techniques have been proposed to improve on the previous method by checking first if a node needs to install a code update. One example is Ripple[16] that reduces the number of messages sent using a publish/subscribe scheme where a single node in the neighborhood sends code updates to its subscribers. Another example is Trickle[10] that periodically broadcasts meta-data about the software version nodes are using. Trickle uses a technique called "polite gossip" to inform neighbors about available software: in each round nodes do not repeat the same version information that they overheard from one of their neighbors. Although both of these approaches do not require all nodes to forward the code update, they still distribute code updates to all nodes that have not installed the new software version yet rather than sending the update to only those nodes that need it for their role. Thus there still is a considerable amount of unnecessary communication.

A better solution would transmit code updates to only those nodes that need it while using only those nodes in between as routers that are necessary to reach all target nodes. For example, in mobile ad hoc networks (MANETs) multicast algorithms such as MAODV[15] could be used for this purpose. For sensor networks with their severe resource limitations there is no such standard algorithm yet. However, since roles are already assigned to the nodes, this information along with knowledge about the network topology can be used to efficiently distribute code updates in the network, as we will see in the next sections.

2.3 On-the-fly Code Update Capabilities

Given the limited energy on sensor nodes, the size of code images, and the energy costs for packet transmission, there should be some way of only updating the parts of the code that change in order to support adaptation and reconfiguration. For adaptation it should be possible to replace parts of the code on a node without having to retransmit them if they have been part of the code image in a previous transmission.

However, most current code update algorithms always transmit the complete code image (including the operating system), which usually amounts to several kilobytes, or blindly divide the code image into blocks without considering the structure of the code. Examples for these two approaches are XNP[7] and Deluge.[5] XNP is included in TinyOS 1.1. It lacks the ability to forward code in a multi-hop network and simply broadcasts the complete code image in a single-hop network. Deluge has been included in more recent TinyOS releases to replace XNP. It allows for incremental updates by dividing the code into fixed-size pages. In addition, it includes functionality to disseminate the update in a multi-hop network while keeping the number of network packets low.

A possible solution to the code update problem is to compare the new code with the previously installed software version and transmit only the differences. Reijers and Langendoen[13] use a diff-like approach to compute an edit script that transforms the installed code image into a new one. Likewise, the incremental network programming protocol presented by Jeong and Culler[6] uses the Rsync algorithm to find variable-sized blocks that exist in both code images and then only transmits the differences. However, both of these approaches just compare the bytes of the code without using knowledge about the application structure. In some cases this will lead to inefficient behavior.

In all of these approaches all nodes will eventually have installed the same code image without support for adaptation. However, in order to reduce the number of packets, it might be desirable to install the required components only on those nodes that need it and store the other ones – if they are received at all – for later adaptation in a free part of flash memory, which typically is less constrained than program memory. Therefore, our solution uses knowledge about the application structure by grouping code into packages of components. It offers more flexibility than simply replacing arbitrary pieces of code because it makes it possible to dynamically change the current set of installed packages through adaptation. That way the sensor nodes can possess several components even though they only need one of them for their current role. When the role changes or other factors make it necessary, the node can easily exchange the currently used component.

3 Network Management in `TinyCubus`

Let us now discuss the specific approaches used in the `TinyCubus` project at the University of Stuttgart to address the challenges of role assignment, code distribution, and code update capabilities.

3.1 Overview of `TinyCubus`

The `TinyCubus`[11, 12] project focuses on configuration management and adaptation issues in sensor networks. The overall architecture of `TinyCubus` has been developed with the goal of creating a generic reconfigurable framework for sensor networks. As shown in Fig. 1, `TinyCubus` is implemented on top of TinyOS[4] using the nesC programming language,[2] which allows for the definition of components in the sense of TinyOS. All applications register their requirements within `TinyCubus` and are executed by the framework.

Although our implementation is based on a TinyOS, `TinyCubus` still is a generic framework since its concepts can also be applied with other underlying operating systems. In addition, TinyOS has been ported to several hardware platforms and has attracted a large community of users so that it can be considered as the de facto standard for sensor network operating systems.

`TinyCubus` itself consists of three parts: the Tiny Data Management Framework, the Tiny Cross-Layer Framework, and the Tiny Configuration Engine. These parts are briefly described in the following sections (refer to our previous work[12] for details).

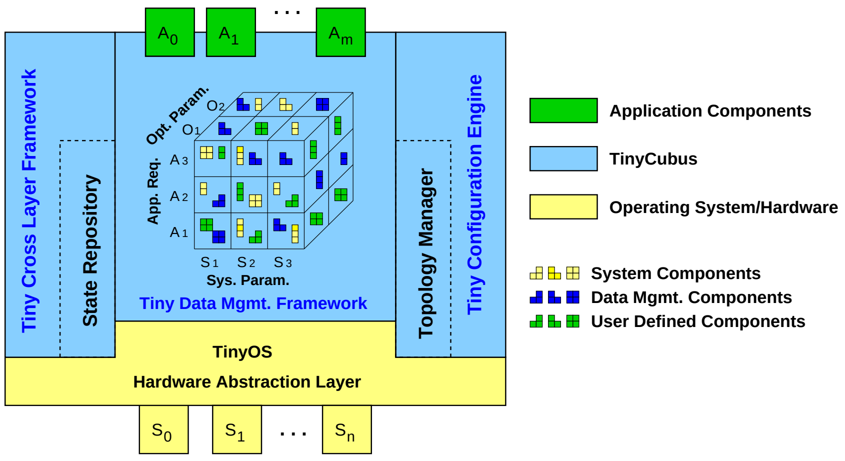

Fig. 1: Architectural components in `TinyCubus`

3.1.1 Tiny Data Management Framework

The Tiny Data Management Framework provides a set of data management and system components. For each type of widely-used data management component such as replication/caching, prefetching/hoarding, aggregation, as well as each type of system component, such as time synchronization and broadcast strategies, it is expected that several implementations exist. The Tiny Data Management Framework is then responsible for the selection of the appropriate implementation based on the current information contained in the system.

The cube in Fig. 1, called 'Cubus', combines *optimization parameters*, such as energy, communication latency or bandwidth; *application requirements*, such as reliability or consistency level;

and *system parameters*, such as mobility or network density. 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 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 dynamically throughout the lifetime of the system and is a crucial part of the optimization process. We are currently investigating different strategies that determine when it is necessary – and beneficial – to select a different component.

3.1.2 Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support the parametrization of components that use cross-layer interactions. As described by Goldsmith and Wicker,[3] strict layering is not practical for wireless sensor networks because it might not be possible then 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. We use a state repository to store the cross-layer data of all components, i.e., the components do not interact directly with each other. Thus architectural properties such as modularity are better preserved than with the unbridled use of cross-layer interactions.

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 also supports this form of interaction. 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.

3.1.3 Tiny Configuration Engine

In some cases parametrization, 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 efficient configuration of both system and application components with the assistance of the 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. In previous work[14] we describe a generic specification language and a role assignment algorithm that are outlined in section 3.2.

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 3.3, this information can be used by the configuration engine to distribute code efficiently in the network.

3.2 Role Assignment for Sensor Networks

3.2.1 Overview

As described above, there is a need for a generic self-configuration algorithm that assigns roles to the sensor nodes. In the case of clustering, one could assign three roles: `CLUSTERHEAD`, `GATEWAY`, and `SLAVE`. In each cluster there is exactly one cluster head, its slaves communicate only with this node, and communication between clusters is routed through gateway nodes. Additionally, roles can be based on the hardware capabilities of a node. For instance, a node with the role `VIBRATION` might have received this role because it is equipped with a vibration sensor whereas the role `TEMPERATURE` characterizes nodes with temperature sensors.

From these examples, we have identified four core elements of systems that support role assignment:[14] a property directory, a role specification, a role assignment algorithm, and some basic services.

The *property directory* contains information about the hardware characteristics of individual nodes as well as dynamic properties such as the remaining battery power or the location of the node. In `TinyCubus` these properties can be retrieved from the cross-layer framework. When they change, role assignment might have to be adjusted dynamically.

For the *role specification*, our topology manager uses a generic specification language. In the specification language a role is defined by a rule. 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 }
```

This is a simplification of a rule that would be used in a real system in which cluster heads may change dynamically. Those rules would also include other parameters such as the amount of energy available or the number of times a node has been cluster head before.

All nodes share the same role specification. Therefore, copies of this specification have to be present on all nodes because our distributed *role assignment algorithm* is executed on each of them. Triggered by property and role changes on nodes in the neighborhood, the algorithm evaluates the rules contained in the role specification. If a rule is satisfied, the associated role is assigned.

Whenever possible, a rule 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 section 3.2.2 for details).

The *basic services* that are needed for role assignment include algorithms for time synchronization, node localization, and neighbor discovery. For this purpose we use existing algorithms that are placed in the data management framework.

3.2.2 Role Assignment Algorithm

The role assignment algorithm works as follows. First, a node evaluating its rules locally broadcasts a `request` message. Since all nodes share the same set of rules, the receivers know what data neighboring nodes need. If a receiving node of such a `request` message is also currently evaluating its rules, conflicting role assignments could arise. For example, two neighboring nodes could send their `request` messages concurrently while they have no role assigned yet. Because they would not detect another `CLUSTERHEAD`, both of them would adopt this role. To avoid these conflicts, receivers of `request` messages also evaluating a rule ensure atomic rule evaluation by either sending back an `abort` message or aborting their own evaluation, depending on their IDs. If there is no such conflict, the receivers locally evaluate the expressions requested and send back the outcome in a `reply` message. If the initiating node receives any `abort` messages, it aborts rule evaluation. Otherwise, it uses the results from the `reply` messages to complete the evaluation and select a role. To allow the neighboring nodes to react by changing their roles, a node broadcasts a `confirm` message to its neighborhood if it changes its role, if there were any `abort` messages, or if properties change. This message contains the new role and information about changed properties. Whenever a node receives a `confirm` message, the receivers start rule evaluation if they were aborted earlier, if the role of the sender has changed, or if some properties of the sender have changed. In addition, rule evaluation is started on nodes that are turned on and on those nodes whose properties change. To reduce the number of `abort` messages, nodes randomly delay the start of rule evaluation.

This algorithm is a first straight-forward solution that has some issues with correctness, robustness, and efficiency.

Regarding correctness the main problem is how to ensure that a stable role assignment has been reached. This means that if there are no property changes, all nodes should eventually be assigned a role that does not lead to role changes of other nodes. Generally, such characteristics depend on the particular role specification. Thus they have to be ensured by the programmer defining the roles. However, in some cases heuristics such as preventing nodes from returning to earlier roles could be applied or the runtime system could detect patterns of illegal behavior. In addition, it must be possible to detect when a stable role assignment has been reached so that the application can react appropriately. For example, a role could be considered stable when it has not changed for a given amount of time.

With respect to robustness, node failures and lost messages are the main issues. If a node fails while the algorithm is outside the role assignment process, in future assignments its absence will be automatically considered because it can no longer report any properties. However, to restart role assignment on the remaining nodes, there must be some way to detect node failures (for example, by periodically re-evaluating rules).

If `reply` messages are lost during role assignment, the properties of the respective nodes are ignored which can lead to the assignment of wrong roles. In addition, if `confirm` messages are lost, aborted nodes do not know when to re-evaluate their rules. For both problems timeouts can be used to ignore late replies and to start the evaluation process again on aborted nodes.

Having in mind the resource constraints of sensor networks, it is essential that the role assignment algorithm works efficiently. Therefore, the message overhead has to be minimized. Knowing that all nodes share the same rules, nodes could forward only the information necessary to re-evaluate the rules on those node affected by a role change. In addition, this data could be cached on the receivers making it unnecessary to re-send it if it does not change. Our algorithm already tries to reduce the number of messages by broadcasting information only locally to the neighborhood. It can be derived from the specification how many hops have to be included in this neighborhood.

In summary, our role assignment algorithm provides a generic solution for self-configuration in sensor networks. However, there are still some issues that need to be addressed before it can be used in real-world deployments.

3.3 Efficient code distribution

In this section we describe our algorithm for code distribution. It makes use of cross-layer information about the topology and the role assignment to efficiently distribute code and other data in a sensor network. We first define the network model used by this algorithm, then describe the algorithm in detail, evaluate its performance, and finally describe its advantages.

3.3.1 Network Model

For our purposes, 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 $A : N \to R$ 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: $k_N = \max\{k_{r_1}, \cdots, k_{r_m}\}$

3.3.2 Detailed Description

Our code distribution algorithm uses the network model presented in the previous section and the information about role assignments provided by the Tiny Cross-Layer Framework to efficiently disseminate code updates to specific roles. This approach is particularly effective if role assignments are stable, for example if roles only depend on fixed hardware properties. However, even if a node's role changes and it does not have the necessary code, the node can request it either from one of its neighbors or from a code repository available outside the network. For the remainder of this paper we assume that role assignments are stable as they are found in the Sustainable Bridges project.[12, 17]

The algorithm starts at gateway nodes by broadcasting data to their 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 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.

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.

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 by Tseng et al.[18] 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.

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 upfront. 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.

3.3.3 Evaluation

In order to show the feasibility of our approach, we have implemented the role-based code distribution algorithm for motes running TinyOS.[4] 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 of these experiments have been obtained using TOSSIM, the TinyOS simulator provided by UC Berkeley.[9]

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.

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. 2 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

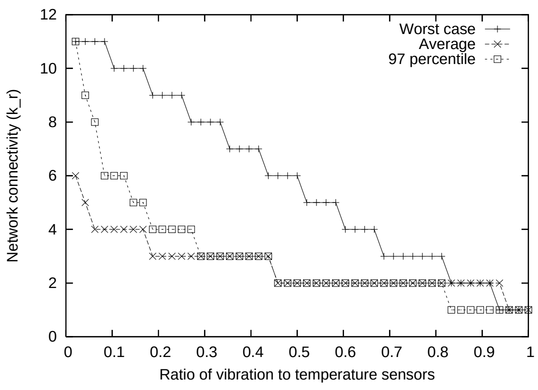

Fig. 2: k_r for 100% completeness (random)

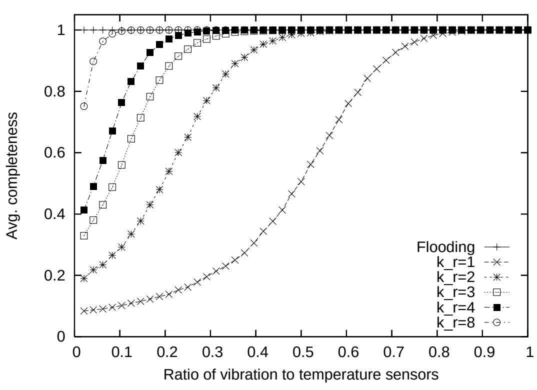

Fig. 3: Avg. completeness for k_r (random)

case indicates a value of at least $k_r = 8$. The curves of Fig. 2 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 $n \leq m$, 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 scenario is:

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

When the algorithm is used for applications other than code distribution, there might, however, be situations where 100% completeness is not required. For these cases, Fig. 3 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. 3 tells us that with $k_r = 4$, we can achieve (on average) the desired results.

Performance Evaluation In the second set of experiments, we have focused on evaluating the performance of our role-based distribution algorithm compared to a flooding approach. We selected flooding as a reference because there is no standard multicast algorithm for sensor networks yet and because such a flooding approach forms the basis of our algorithm. However, our approach could also be combined with more advanced distribution schemes such as Trickle[10] (see section 2.2), which dynamically selects the nodes forwarding a code update.

For these experiments, the sensor nodes are laid out in an evenly spaced $m \times n = 12 \times 4$ grid with a rectangle of vibration sensors enclosing some temperature nodes. 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. We focus on this structured grid topology with well-known roles assigned to the nodes because it represents the topology of the Sustainable Bridges application.[12]

In our simulations, 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.

Fig. 4 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 described above to all nodes being assigned the `VIBRATION` role. The measurements shown are the average of 100 runs. In the graph,

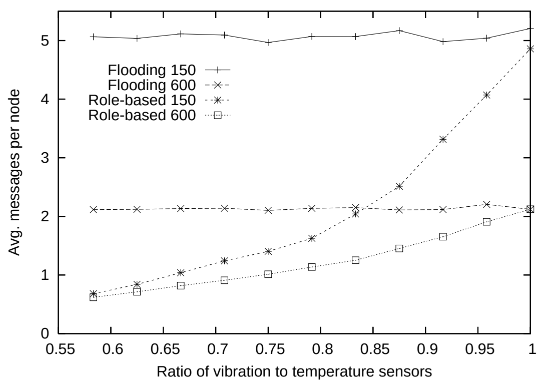

Fig. 4: Avg. number of messages/node (structured)

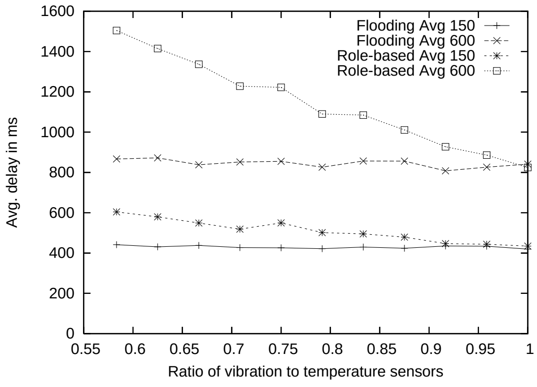

Fig. 5: Avg. delay (structured)

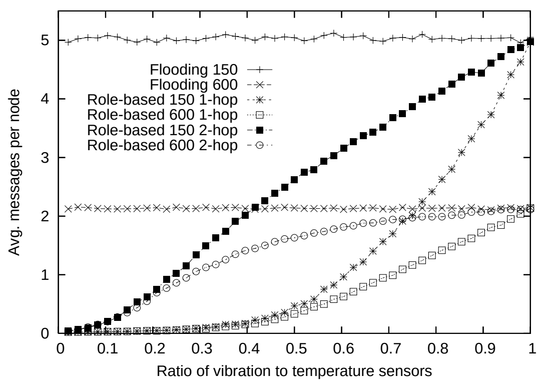

Fig. 6: Avg. number of messages/node (random)

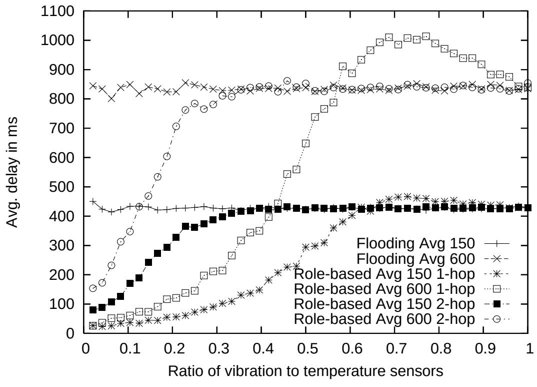

Fig. 7: Avg. delay (random)

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 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 between 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.[*] As expected, the number of messages per node increases as the ratio of vibration to temperature sensors increases. In the extreme case (when all nodes in the network are vibration nodes), our algorithm behaves just like flooding.

Fig. 5 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. 4), while achieving delays just slightly above those of flooding (compare Flooding Avg 150 and Role-based Avg 150 in Fig. 5).

As we have just seen, our role-based distribution algorithm can be used very efficiently with

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

structured topologies 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. 6 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. 3, 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. In contrast, if the value chosen for k_r is greater than the actual connectivity of the network, unnecessary messages are sent which makes the algorithm behave more like flooding. This is the reason why in Fig. 6 the slopes of the curves for the 1-hop algorithm are greater than those for the 2-hop algorithm, especially near 1.0 where (according to Fig. 3) the 2-hop version probabilistically reaches all nodes.

Finally, analogously to the structured case, Fig. 7 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, our algorithm has lower delays than flooding for low ratios of vibration sensors but, as shown in Fig. 3, 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, see Fig. 3), 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. 5. 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, thus, 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. 5 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.

3.3.4 Advantages of Role-Based Code Distribution

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 approaches.

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 here 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.

3.4 On-the-fly Code Update

Let us now describe the techniques used in `TinyCubus` to tackle the efficient on-the-fly code update problem. We call our approach *Flexible Code Update (FlexCUP)* because besides updating the code image it adds the flexibility needed to support adaptation.

3.4.1 Flexible Code Updates

In our chosen implementation platform – TinyOS-based motes – dynamically updating parts of the code image is very challenging. The reason is that the nesC compiler statically combines both application and system code into a single binary object. Since the separation of nesC components is not preserved in the compiled code, there is no way to distinguish different modules in their binary representation. In addition, memory is statically allocated and cannot be reallocated at runtime.

In order to allow for reallocation of functions and global variables, the nesC compiler needs to be modified. Normally, the nesC compiler creates one large C file that contains the complete source code of all system and application components needed by the application. This file is later compiled by a platform-specific version of the GNU C compiler (AVR-GCC). With *FlexCUP* we break up such a C file into separate files (called “packages”), each of which contains some logically associated nesC components. We then compile these packages separately and link them either on a PC or – to support the exchange of individual packages – on the motes.

Each of the packages can be exchanged individually, given that the new one provides the same interfaces and has the same external references as the previous one. However, when a new version of a package is installed, some addresses need to be patched, both within the new package and within the rest of the application. The reason for this is that the position of functions and variables in memory may change because the size of functions and of variables might change from version to version. Thus, besides the mere code some additional information (metadata) has to be sent to the nodes in order to perform these changes. Using this metadata, linking can be performed on the motes instead of on a PC.

By exchanging packages instead of arbitrary blocks, the update process follows the natural view of the programmer. Since each package contains code that logically belongs together, we expect that changes in an application are often local to a single package. On the motes only some references to functions have to be adjusted. In contrast, if there is a change in one part of the program that leads to address shifts in other parts, block-based algorithms would transfer a block – in some cases with a fixed size – for each single change of an address. Obviously, this is very inefficient because in many cases almost the complete code image has to be transmitted.

With our approach, the Tiny Data-Management Framework (see section 3.1.1) becomes feasible. Given that complete packages with associated metadata for the same component type exist, they can be dynamically exchanged during the adaptation process without any intervention from outside the node. This flexibility is a key advantage of our solution compared to protocols that compute the difference of two code images without considering the structure of the code.

With *FlexCUP* we create separate object files for each package. If just one package needs to be replaced, only the corresponding object file has to be transmitted. The data needed on the motes to integrate this file into the application is the code itself, the symbol table, and the relocation table. The symbol table contains an id for each symbol and its address within the current package. Similarly, the relocation table contains an entry for each access to such a symbol. We reduce the size of these tables by inlining most internal functions within a package. In addition, we compress the size of the relocation table by combining similar entries. For example, if there are several relocation entries for a symbol, they are grouped and the identifier of the symbol is given only once. Furthermore, we plan to strip function symbols that are only used locally from the symbol table to further reduce its size.

The relocation table for storage access is more complex. Because global variables can be accessed by any package and because variables are not grouped in memory regarding the package they are contained in, it is not possible to distinguish between variables internal to a package and those that are accessed externally. Therefore, the identifier of the symbol is always required. In addition, there are several types of such relocations that differ in the way addresses are stored. Similar to the relocation table for function symbols, we group entries based on their type, referenced symbol, and the offsets of the command accessing memory within the package.

Furthermore, we try to minimize the effects of changes by placing the packages intelligently in memory. For example, the system package is accessed by many other ones and is less likely to change. Therefore, it is placed at the first position in memory. When an application-specific package is updated and changes its size, the system package and all references to it do not have to be modified.

Finally, if the exact version installed on the sensor nodes is known in advance, this basic scheme (called *FlexCUP Basic*) can be further optimized by combining it with Reijers's and Langendoen's diff-based algorithm leading to *FlexCUP Diff*. This way only the differences between the new code

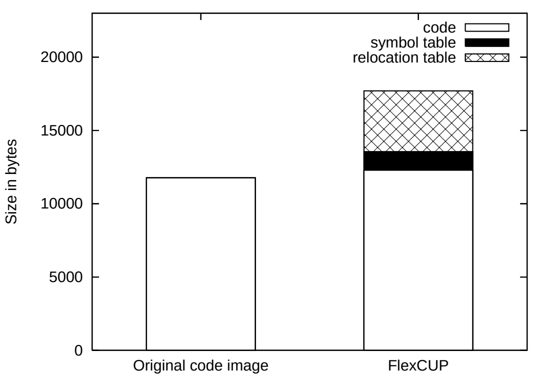

Fig. 8: Size of the original and the packetized OscilloscopeRF application on the motes

package and the previous one need to be transmitted. The computation of these differences is easier in our approach because addresses are set to zero by the compiler and inserted later when the package is linked on the motes. Since changed addresses are one of the main things that need to be fixed, *FlexCUP Diff* produces edit scripts with fewer entries than the pure diff-based algorithm.

3.4.2 Evaluation

We evaluated *FlexCUP* using the OscilloscopeRF application, which is included in the TinyOS distribution. This sample application periodically reads sensor data from the light sensor and transmits it to a base station using a radio link. On a PC an oscilloscope-like Java application can be used to visualize the sensor readings. We divided the OscilloscopeRF application into five packages of components that belong together. We created a radio package, a sensor package, a timer package, an application package, and a system package. The radio package contains all components associated with radio communication and the sensor package all those used to access the light sensor. The timer package includes the components that provide the timing service. The application package contains the actual application components and the main functions. Finally, the system package contains hardware-specific components as well as other basic functions provided by the operating system.

Usually, the nesC compiler makes heavy use of inlining to optimize the applications. With *FlexCUP* this is still possible within a package. However, functions called from another package cannot be inlined. Therefore, the stack consumes more memory at runtime. In addition, the size of the code image increases slightly[2] because the steps needed to call a small function prevail over the code in the function itself. For the OscilloscopeRF application the size of the program image increases from 11,768 bytes to 12,284 bytes (about 5%) for the five packages described above (see Fig. 8). In addition, in order to be able to link applications on the motes, for this application 5,417 bytes of metadata have to be stored on the motes. 1,273 bytes of this metadata are used for the symbol table and 4,144 bytes for the relocation tables (mostly for storage accesses). Compared to the size of the application the size of the metadata seems to be quite large. However, if the flexibility gained because of this data and the reduced size of code updates is considered, this amount of metadata is acceptable. We are also exploring ways to further reduce the size of the tables stored on the motes and transmitted using radio communication. For example, a diff-based approach could also be applied to transfer the symbol and relocation tables. Note that this metadata does not have to be stored in RAM, which is very limited on the sensor nodes. Instead, it can be kept in flash memory and only be loaded when it is necessary for the installation of a new component.

To evaluate the effectiveness of *FlexCUP* when code changes, we performed several typical changes to the application and the sensor package. First, we made a minor change by calling one function during initialization twice. There is no real benefit for this modification but it shows how much network traffic a small bugfix would require. After this small change the size of the application package increased from 436 to 440 bytes. For our approach we also have to transmit the symbol and relocation table of this package (328 bytes). Thus with *FlexCUP Basic* the update requires 768

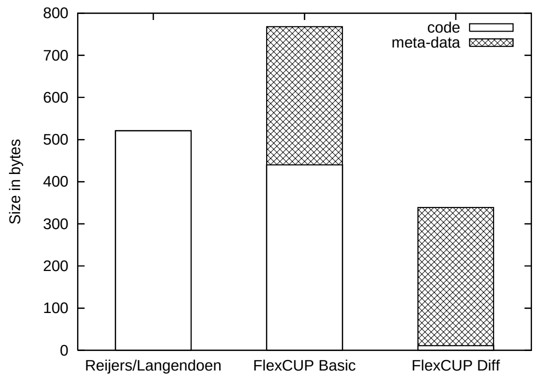

Fig. 9: Bytes to transfer for a simple bug fix

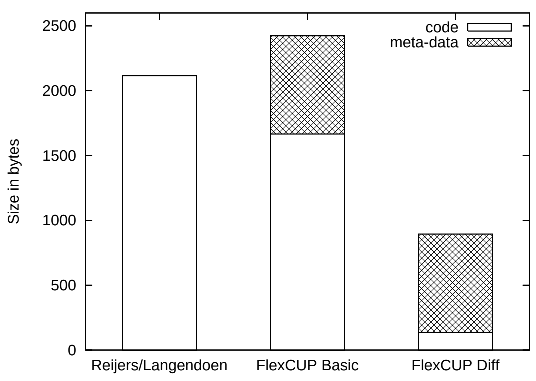

Fig. 10: Bytes to transfer for a more complex change

bytes in total (see Fig. 9). In contrast, an implementation of Reijers's and Langendoen's diff-based approach for Mica2 motes[19] transmits only 521 bytes for this change. However, if we use *FlexCUP Diff* and apply their algorithm on just the package that changes, our approach sends only 11 bytes for the code along with the 328 bytes for the metadata, i.e., 339 bytes in total. This is 35% less than the pure diff-based approach because addresses are not inserted in the code yet. Thus fewer changes are necessary. In comparison, approaches that always transmit the complete code image would have to send almost 12 kilobytes of code.

We then made a more complex change by adding code to the light sensor that computes a moving average over the last four sensor readings. This introduces three additional global variables, adds five lines of code, and modifies another one. After this modification the size of the sensor package increased from 1,594 bytes to 1,666 bytes. The size of the corresponding metadata is 758 bytes so that, in sum, *FlexCUP Basic* transmits 2,424 bytes. As shown in Fig. 10, Reijers's and Langendoen's pure diff-based algorithm sends 2,116 bytes and *FlexCUP Diff* only 895 bytes.

The results of this evaluation are very promising. Although in some cases *FlexCUP Basic* transmits more bytes than Reijers's and Langendoen's diff-based algorithm, we think that the overhead is still acceptable considering the flexibility gained through our approach. In addition, by combining both approaches as done with *FlexCUP Diff* the overhead can be further reduced so that this combination produces better results than the pure diff-based solution and still preserves the possibility to perform adaptation on the motes.

4 Conclusion and Future Work

With the increasing complexity of sensor networks and sensor network applications, management and configuration techniques are clearly needed. We have identified three of the major research problems in this field and briefly described our solutions for each of them. The first challenge is the assignment of tasks to the sensor nodes that we address with our specification language and our role assignment algorithm that allows the network to dynamically adjust to the current situation. We have described how the algorithm works and have shown some possibilities to further improve it. The second research problem is the efficient distribution of code updates in the network. We have presented and evaluated an algorithm that leverages the role information to distribute code updates – or other data – in the network. Finally, the third challenge is the reduction of the size of code updates and adding flexibility by supporting adaptation. In our approach called *FlexCUP* we divide the code image into separately updateable packages and adjust references within the code on the motes. Techniques for configuration management in sensor networks have to be efficient, practical, and energy-aware. As we have shown, we are developing such technology in the `TinyCubus` project.

However, there is still some work to do. One challenge is the development of an efficient distributed role assignment algorithm that is resilient to node failures. In addition, we plan to extend the role-based code distribution algorithm to support highly mobile sensor nodes and to add functionality found in related projects such as Trickle. Regarding code updates, we are working on reducing the overhead of metadata required on the motes and on increasing the speed of component transmission so that on-the-fly adaptation is possible. Finally, the implementation of `TinyCubus` is still under way with prototypes of the cross-layer framework and configuration engine being already partially functional.

References

[1] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In F. Zhao and L. J. Guibas, editors, *Proc. of the 2nd Intl. Workshop on Information Processing in Sensor Networks*, pages 47–62, 2003.

[2] 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*, pages 1–11, 2003.

[3] A. J. Goldsmith and S. B. Wicker. Design challenges for energy-constrained ad hoc wireless networks. *IEEE Wireless Communications*, 9(4):8–27, 2002.

[4] 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*, pages 93–104, 2000.

[5] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd Intl. Conf. on Embedded Networked Sensor Systems*, pages 81–94, 2004.

[6] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *First IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks*, 2004.

[7] J. Jeong, S. Kim, and A. Broad. Network reprogramming, 2003. `http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf`. Accessed on 12/17/04.

[8] M. Kochhal, L. Schwiebert, and S. Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Appl.*, pages 98–107, 2003.

[9] 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*, pages 126–137, 2003.

[10] 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.

[11] 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*, pages 623–628, 2004.

[12] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proc. of the 2nd European Workshop on Wireless Sensor Networks*, pages 278–289, 2005.

[13] 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.*, pages 60–67, 2003.

[14] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *Proc of the 11th ACM SIGOPS European Workshop*, pages 7–12, 2004.

[15] E. M. Royer and C. E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *Proc. of the Intl. Conf. on Mobile Computing and Netorking.*, pages 207–218, 1999.

[16] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, L.A., 2003.

[17] Sustainable bridges web site. `http://www.sustainablebridges.net`. Accessed on 12/17/04.

[18] 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*, 8(2/3):153–167, 2002.

[19] T. Yeh, H. Yamamoto, and T. Stathopolous. Over-the-air reprogramming of wireless sensor nodes. UCLA EE202A Project Report, 2003. `http://lecs.cs.ucla.edu/~thanos/EE202a_final_writeup.pdf`. Accessed on 12/21/04.