TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks

TinyCubus: Ein Adaptives Cross-Layer Framework für Sensornetze

Pedro José Marrón, Daniel Minder, Andreas Lachenmann, Kurt Rothermel, Universität Stuttgart

Summary 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 cross-layer framework that enables optimizations through cross-layer interactions, a configuration engine that distributes components efficiently by considering the roles of the sensor nodes and provides support to install components dynamically, and a data management framework that selects and adapts both system and data management components. Finally, relevant research challenges associated with the development of each framework are identified and discussed in the paper. ►►►

Zusammenfassung Mit der zunehmenden Verbreitung von Sensornetzen und Sensornetzanwendungen wächst die Komplexität solcher Systeme ständig an. Sensornetze sind nun in Bezug auf ihre Hardware-Eigenschaften und Anwendungsanforderungen häufig sogar in einem einzelnen Netz heterogen. Außerdem wird erwartet, dass die Anforderungen der Anwendungen sich mit der Zeit ändern. Dies alles erschwert es, Sensornetzanwendungen zu entwickeln, einzusetzen und zu optimieren. In diesem Artikel stellen wir die Architektur von `TinyCubus` vor, einem flexiblen und adaptiven Cross-Layer Framework für TinyOS-basierte Sensornetze, dessen Ziel es ist, die notwendige Infrastruktur bereitzustellen, um die Komplexität solcher Systeme zu unterstützen. `TinyCubus` besteht aus einem Cross-Layer-Framework, das Optimierungen durch Cross-Layer-Interaktionen ermöglicht, einer Configuration-Engine, die Komponenten effizient durch die Beachtung der Rollen der Sensorknoten verteilt und die dynamische Installation von Komponenten erlaubt, und einem Data-Management-Framework, das sowohl System- als auch Datenverwaltungskomponenten dynamisch auswählt und adaptiert. Schließlich werden in diesem Beitrag relevante Forschungsprobleme, die die Entwicklung der einzelnen Teile betreffen, bestimmt und diskutiert.

KEYWORDS C.2.1 [Network Architecture and Design – Wireless communication], C.2.3 [Network Operations – Network management], D.4.7 [Organization and Design – Distributed systems], sensor networks, system architecture, adaptation, cross-layer

1 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 (also called nodes) that are equipped with sensors. Each node is able to process data in the network and transmit it using multi-hop communication. Furthermore, most nodes are resource-constrained so that energy consumption plays an important role. Additionally, depending on the field of application,

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 [6], there is no standard platform for sensor nodes yet. Even different models of motes running TinyOS differ greatly and new hardware is continuously being developed. Likewise, applications are rapidly 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 [16], whereas others are developed for military operations, medical care or rescue operations.

Finally, the sensor network itself might also be heterogeneous: In current applications, a network often contains devices with different functionality. 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. If these requirements change or another application is executed, the network itself has to adapt, and this is no easy task. All of this makes developing, optimizing, and deploying sensor network applications a complex and error-prone task. Therefore, in order to simplify adaptive 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` [10; 11], which aims at providing the necessary infrastructure to support the complexity of such systems. `TinyCubus` consists of a cross-layer framework, a configuration engine, and a data management framework. 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 *data management framework* supports the dynamic selection and adaptation of system and data management components.

The remainder of this paper is structured as follows. The next section describes the requirements of two specific sensor network applications. Section 3 presents the overall architecture of our framework and gives more detailed information about its three parts and the research challenges associated with their development and use. Furthermore, in this section we describe and evaluate the scheme used by the configuration engine to distribute code updates in the network. Finally, Section 4 gives an overview of related work and Section 5 concludes this paper and describes future directions.

2 Application Requirements

At the University of Stuttgart we are working on two sensor network projects that act as canonical examples for the study of static and mobile sensor node applications: *Sustainable Bridges* [15] and *Cartalk 2000* [12].

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. A wide range of sensor data such as temperature, humidity, vibration, as well as noice detection and localization mechanisms are needed to achieve this goal. In order to determine the position of cracks, noise emitted by the bridge is sampled and, by using triangulation methods, the position of the possible defect is determined.

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 move relative to each other and, therefore, algorithms that are able to cope with mobile sensors are needed to accurately process data.

2.1 Application Comparison

Obviously, the *Sustainable Bridges* and *Cartalk* 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

Table 1 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	◔	●
Time Sync	◕	◔
Topology	●	◑

○ Not important ◑ Medium ● Very important

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 both applications perform sensitive monitoring tasks, they 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.

However, these applications also have considerable differences. Table 1 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 data model, whereas *Cartalk* needs a generic data model to support generic user interaction. Regarding the query model, in 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 *Cartalk* need to be able to specify their own queries. Therefore, *Cartalk* mostly requires a pull-based (query-based) mechanism. In this application only the data and not the node id is important, whereas in *Sustainable Bridges* the exact source node of the data has to be known, and so there is no need for distribution transparency in this application. Moreover, energy constraints are only important for *Sustainable Bridges* and mobile nodes only exist within *Cartalk*. *Sustainable Bridges*, on the other hand, has very strict time synchronization requirements to ensure good event localization quality, but for *Cartalk* less accurate synchronization is sufficient. Finally, regarding topological constraints, *Sustainable Bridges* 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* topological information is limited to the use of road and city maps.

2.2 Requirements for a Generic Framework

Despite all these differences, our exemplary applications obviously have some commonalities. Therefore, it is possible to simplify the development of both applications – and of others that share some properties with them – by creating a generic framework for sensor network applications.

Such a framework has to provide the common functionality required by a broad class of applications. It 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. Such a framework must then be able to *adapt* to these conditions and *support optimizations*, especially because of the resource limitations found in sensor networks.

The use of a generic framework or of several more specific frameworks for different classes of applications are two possible equivalent solutions that implement these requirements. In this paper, however, we present the architecture

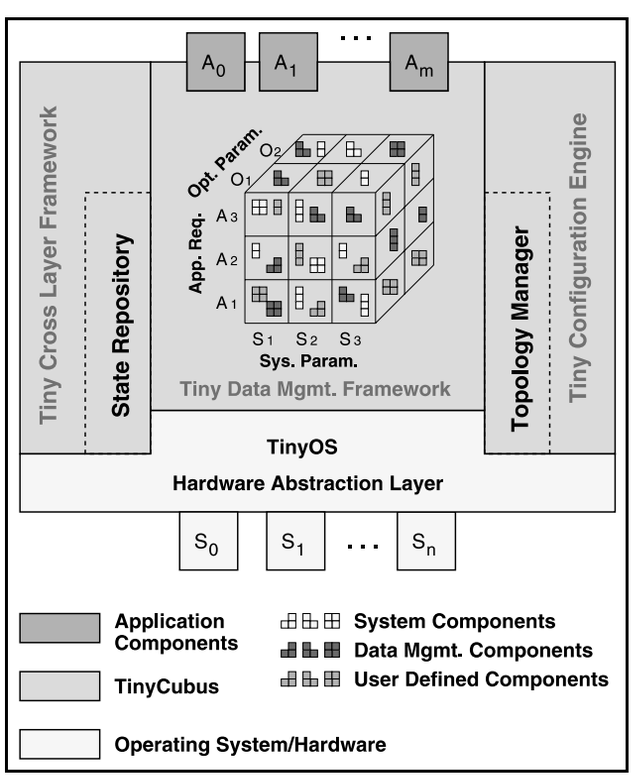

Figure 1 TinyCubus architecture.

of a generic framework, since its internal structure is the same independently of whether or not it is intended for all classes or just a certain number of applications.

3 TinyCubus 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 Fig. 1, `TinyCubus` is implemented on top of TinyOS [6] using the nesC programming language [3], 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 Cross-Layer Framework*, the *Tiny Configuration Engine*, and the *Tiny Data Management Framework*, which are described in the following sections.

3.1 Tiny Cross-Layer Framework

The goal of the *Tiny Cross-Layer Framework* is to provide 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 [5] 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. The second form of cross-layer interactions has to do with the execution of custom, application-specific code via callbacks to higher-level components. For example, an application-specific data analysis function such as a fast Fourier transform (FFT) can be invoked whenever new sensor data is available.

The *Tiny Cross-Layer Framework* provides support for both parameter definition and custom code execution. Our framework 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 custom code, the cross-layer framework makes use of `TinyCubus`′ ability to execute dynamically loaded code.

3.1.1 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 changing 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. In fact, the new component can build on the configuration of the old one to make the transition smoother. 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 that allows us to specify what cross-layer data a component needs and provides. From this specification the system generates an interface that can be used to access the data with type safety. In contrast to other TinyOS interfaces, this interface is not wired to a specific implementation but rather to the data itself. So the developer of a component does not have to know which component actually provides the cross-layer data. Whenever the data management framework selects a different component, the wiring of the interface is adjusted to the new component. In addition, with this specification components that make cross-layer data available can also determine if other ones use it and if they have to gather data at all.

Currently, we have implemented a very basic version of the state repository that supports data sharing between components although it does not include the specification language and support for adaptation yet. We use this simplified cross-layer framework to refine the requirements for our final implementation and to get a better understanding of cross-layer interactions in general.

3.1.2. Custom Code One commonly used approach for cross-layer optimizations is to merge conceptually separate functionality into one layer or component. For instance, the routing and application layers are sometimes implemented as a single component. However, these kinds of optimizations hinder the modularity of applications. In addition, they would prevent the data management framework from exchanging just the routing component, for example. Therefore, the interaction of components has to be reduced to interactions through their interfaces.

In a layered architecture, if layers not directly adjacent interact, all layers in between have to extend their interface to relay this interaction. However, narrow interfaces have the advantage that they are very simple: they only provide methods that are actually needed for a functional requirement. Therefore, our approach does not extend the interface of all components between two interacting ones. Instead, we provide support for the execution of

application-specific code in lower-layer components via callbacks.

TinyOS already provides some support for callbacks 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. As explained in Section 3.2, with TinyCubus components can be exchanged at runtime. Therefore, both the usage of a component and callbacks have to be directed to the new component if a component is replaced. Therefore, in our system interfaces are not wired to specific components. Instead, they are wired to a component type such as "Routing component".

With TinyOS it is possible to wire several components to a single interface. This is also true for our usage of callbacks for the execution of custom code. However, in many cases it might not be desirable to wire more than one component to a callback. For example, if the called-back component modifies a network packet, it expects its modifications not to interfere with other components responding to the same callback. One possible solution to this problem is to limit the number of components wired to a callback interface if there might be conflicts otherwise. If several callbacks do not interfere, they could also be called serially.

3.1.3. Research Challenges There are numerous research challenges associated with the design and implementation of the cross-layer framework. First of all, it is still an open question how a broad variety of cross-layer optimizations can be supported without loosing modularity and other beneficial properties of the software architecture. Furthermore, the effects on the whole system have to be studied when some components are optimized. We address this problem of having components with different optimization goals by selecting only those with compatible optimization parameters in the data management framework. Nevertheless, with cross-layer approaches small changes of parameter values in one component can cascade to a series of changes affecting a large number of components. These effects have to be investigated in more detail.

Regarding the state repository, one challenge is the evaluation of an appropriate data access paradigm (e. g., publish/subscribe or query/response). This issue probably depends on the data being accessed – or even the component using the data. Furthermore, strategies for the storage of state have to be investigated. Especially, these strategies have to consider the resource limitations characteristic for sensor networks.

Furthermore, we have to analyze what types of parameters stored in the state repository exist and how they affect the components. For example, there are simple parameters that just modify some properties of an algorithm that can be easily computed using the parameter (e. g., the signal strength of a networking component). In addition, there are other parameters that affect the structure of an algorithm. They change the execution paths that are selected if there are conditional statements that only depend on cross-layer data. If there are those parameters, the algorithm could be split into several distinct components that only work with some parameter values. Most of the time only one of those smaller components would be needed for an application. A research challenge here is how this division of a component can be supported by the framework, e. g., using code analysis techniques at compile time.

The main challenge concerning the support of callbacks is to create an efficient mechanism for callbacks to and from dynamically selected components. Since callbacks are expected to be used during time-critical operations, the overhead of directing the callback to the right component has to be small. This issue is aggravated by the fact that components can be exchanged at run-time which requires an additional indirection step.

3.2 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 configuration of both system and application components using cross-layer information about the functionality assigned to the nodes.

The configuration engine consists of the topology manager and a code distribution facility. The topology manager enables self-configuration of the network by assigning a role to each node that depends on its functionality. Our code distribution facility installs components dynamically on the nodes. It uses the role information to distribute them efficiently in the network.

3.2.1. 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.

For role assignment the topology manager uses a generic specification language and a decentralized role assignment algorithm [14].

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 [14] for details).

3.2.2. Code Distribution Most existing approaches that distribute code in sensor networks do it by replacing the complete code image. However, most of the time only a single component needs to be updated or replaced. In these cases replacing the complete code image requires the unnecessary transmission of many packets, thus wasting energy. Our configuration engine only transmits the components that have changed and integrates them with the existing code. Of course, when installing a new component, the configuration engine has to make sure that all dependencies are fulfilled.

Since energy is a very limited resource on sensor nodes, the configuration engine tries to reduce the number of packets for code updates even more. Our code distribution algorithm leverages application knowledge about the specific role assigned to each node and sends code updates only to those nodes that belong to a given role and need this code update.

Moreover, the configuration engine takes care of ensuring the reliability of code transmissions by implementing two services, the reliable transmission of individual code fragments and the atomicity of code updates within a node. This is crucial for the proper update of system components such as routing protocols that other higher level components depend on.

In Section 3.2.4 we describe our role-based code distribution algorithm in more detail and present some results of its evaluation.

3.2.3. Research Challenges There are several research challenges associated with the configuration engine. One of them is the definition of the specification language for role assignment and reconfiguration that is integrated in the nesC programming language. Furthermore, roles have to be assigned efficiently because of the resource constraints of sensor networks.

Another research challenge deals with efficient distribution strategies for code updates. Our approach using information about the role assignment certainly is a viable solution. However, it is only suited for sensor networks with certain properties. For instance, it assumes that nodes are stationary. In addition, there is a need for synchronized reconfiguration algorithms. These algorithms ensure that all nodes switch simultaneously to a new transmission protocol, for example.

Furthermore, it is still an unsolved problem how components can be installed dynamically on the sensor nodes. When only some parts of the code are replaced, function calls have to be redirected. There are several possibilities to solve this problem. For example, function calls could be dynamically mapped to the address of the currently used implementation. Alternatively, all calls of this function could be rewritten in program memory.

In hybrid network architectures that consist of both sensor nodes and more powerful infrastructure-based nodes, a code repository can be created on the infrastructure-based nodes. Whenever a sensor node needs a new component, it can be retrieved from this repository. However, it is still unclear how the code repository can be managed efficiently. For example, it is an open question how the code available in the repository should be included in the adaptation decisions of the data management framework. One possible solution is that the data management framework preferably selects components that already have been installed on the nodes in order to reduce the network traffic for code updates.

3.2.4. Details of Role-Based Code Distribution In previous work [10; 11] we describe and evaluate an algorithm that efficiently distributes code updates by sending them only to those nodes that need it. This algorithm leverages cross-layer data in the form of information about the role assigned to a node to determine which nodes actually need the code update.

With this algorithm gateway nodes broadcast data to their k_r-hop neighborhood, where r is a role and k_r is a parameter that specifies the number of hops the algorithm can tolerate over nodes with a different role from r. Then nodes forward such a message to their own k_r-hop neighborhood only if they are assigned role r, thus flooding the nodes with role r while using only those nodes with other roles that are necessary to reach them. It is possible to parametrize the algorithm by selecting k_r for each role, adapting to the topology of the network.

In order to reduce the number of collisions in dense networks each node waits for a random time $t \in [0, \ldots, t_{max}]$ before retransmitting a message. The choice of t_{max} is related to the delay observed in the evaluation. In addition, to deliver code updates reliably our

algorithm uses implicit acknowledgments, i. e., it treats the forwarding of the message by a neighbor as an acknowledgment. If after a certain amount of time a neighboring node has not forwarded the message, the node waiting for the acknowledgment retransmits it. Both of these components can be replaced with any other scheme for collision avoidance and reliability, respectively.

We have implemented this role-based code distribution algorithm for motes running TinyOS. In our experiments, we compare our approach with a flooding algorithm that has been augmented with the same mechanisms as our algorithm to provide reliability and collision avoidance. The results presented here have been obtained using TOSSIM, the TinyOS simulator provided by UC Berkeley [8].

For our experiments we analyzed a scenario from the *Sustainable Bridges* application: Nodes are placed in an evenly spaced 12×4 grid with a rectangle of vibration sensors enclosing some temperature nodes. The sensor nodes are assigned either the `VIBRATION` or the `TEMPERATURE` role. By having each `VIBRATION` sensor forward a message to its 1-hop neighborhood, all `VIBRATION` nodes can be reached. There is a single gateway located in one of the corners. The distance between the nodes is 10 meters and the radio model is set to a lossless disc model with a communication range of 15 meters. We simulated the algorithms with a maximum transmission delay t_{max} of both 150 ms and 600 ms.

Fig. 2 depicts the number of messages sent on average by each node. The graph shows the number of messages sent by both flooding and our role-based code distribution algorithm. Role assignments on the x-axis vary from the original configuration described above to all nodes being assigned the `VIBRATION` role. The values shown are the average of 100 runs.

With the flooding algorithm the average number of messages sent is about 5 message per node for $t_{max} = 150$ ms and a little more than 2 for $t_{max} = 600$ ms. These values are greater than one and depend on t_{max} because the algorithm retransmits messages until all `VIBRATION` nodes are reached. As the graph shows, the number of messages sent with flooding is independent of the role assignment because this protocol does not consider the roles of nodes when distributing data.

The performance of our role-based algorithm is much better than flooding, particularly when the ratio of vibration to temperature sensors is low because only vibration nodes forward the messages. When this ratio is increased, the algorithm behaves more and more like flooding.

Fig. 3 shows the time needed until all vibration nodes are reached. The average delays needed by our role-based algorithm are at most 1.5 times worse than those of flooding since flooding uses more nodes to forward the data in parallel. Because of the topology of our network a single node with a long delay can slow down the complete algorithm. Nevertheless, by setting $t_{max} = 150$ ms it is possible to keep

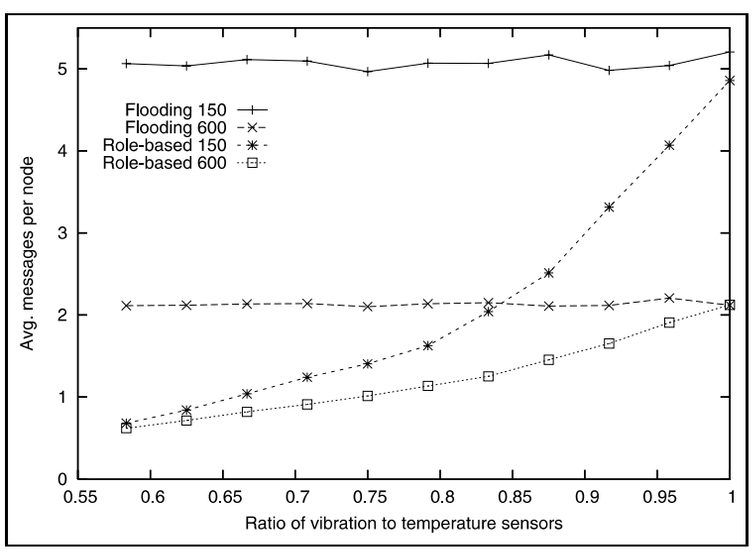

Figure 2 Average number of sent messages per node.

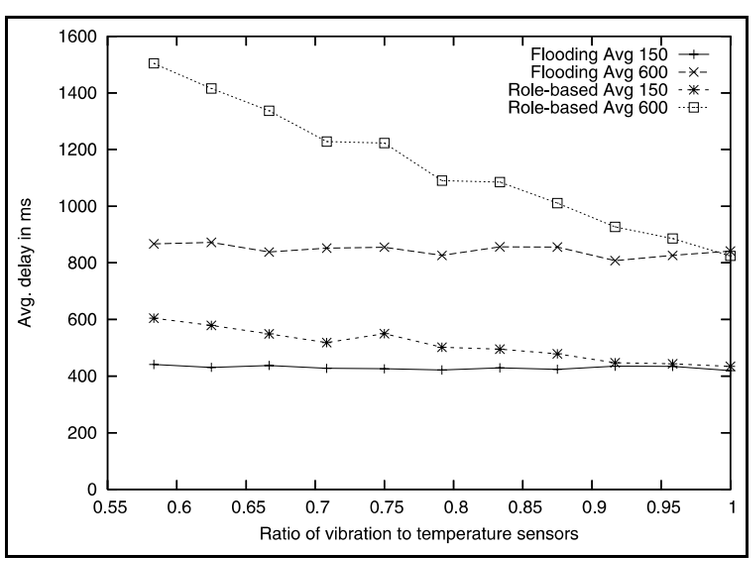

Figure 3 Average delay for message delivery.

the number of sent messages low while achieving delays just slightly above those of flooding.

3.3 Tiny Data Management Framework

The goal of the *Tiny Data Management Framework* is to provide a set of standard data management and system components and to choose the best set of components based on three dimensions, namely system parameters, application requirements, and optimization parameters.

The cube of Fig. 1, called '*Cubus*', represents the conceptual management structure of the *Tiny Data Management Framework*. It is motivated by the experience in developing sensor network applications. When looking for a suitable algorithm, at first, a developer has to pay attention to the characteristics which are exhibited by the environment of a sensor node, e. g., density or mobility of the network. These influencing factors are called *system parameters*. Secondly, application requirements, such as reliability requirements, additionally restrict the set of possible algorithms. Finally, the algorithm is selected that fulfills best some optimization criteria, e. g., minimal energy consumption.

Each dimension itself consists of several parameters as can be seen from the previous 'system parameters' example. Since parameters are mostly independent, every combination (i. e., their power set) has to be considered in each dimension. The number of parameters plays, therefore, a crucial role for the expressiveness as well as the scalability properties of our adaptation algorithm.

For a sensor network application, different types of algorithms are needed. We subdivide them into system algorithms, which are responsible for basic operations such as broadcast or time sychronization, and data management algorithms, which deal mainly with data handling such as replication/caching, prefetching/hoarding, or aggregation. For each type, several algorithms with different properties exist. Using our *Cubus* model, each algorithm is classified according to the three dimensions specified above. For example, a tree based routing algorithm is energy-efficient, but cannot be used in highly mobile scenarios with high reliability requirements. The *Cubus*, then, allows for a reference frame definition where different algorithms can be easily compared.

Each algorithm is implemented as a component suitable for TinyOS and `TinyCubus`. The type of the component is the same as the algorithm's type (i. e., broadcast, time synchronization, or aggregation, for example). Each component is tagged with the classification, i. e., the combination of parameters and requirements in the *Cubus*, of the algorithm it is implementing. Eventually, for each combination a component will be available for each type of data management and system components. Of course, a component can be appropriate for more than one parameter combination. If the reason for this is a different parametrization of the component, it can be split into smaller components as explained in Section 3.1.3.

The *Tiny Data Management Framework* selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. In this set, components for each type of needed functionality are included. For different requirements in one or more dimensions different components of each type may be more appropriate and will, therefore, be selected so that the system as a whole adapts to changing requirements. 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.

3.3.1. Research challenges As indicated before, the parameters and requirements in the three dimensions of the *Cubus* (system parameters, application requirements, and optimization parameters) have to be carefully selected. If too few parameters are chosen, it is possible that more than one component of a single type qualifies for the same set of parameters. Thus, they cannot be distinguished in the framework anymore and additional selection strategies have to be applied. Too many parameters reduce the chance to find a suitable algorithm for every combination of parameters. There would be large holes in the *Cubus*, thus reducing the quality of adaptation.

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 adding more system capabilities, e. g., a positioning system or a monitoring component, additional possibilities arise, such as using information about the spatial extension of the network or the total number of nodes in the network. However, these parameters are incorporated into the framework only if they help distinguish available components. By examining sensor network applications as outlined in Section 2, we determine the *application requirements*. In the broadest sense, they can be subsumed under the term 'quality of service'. Examples are data consistency level, 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.

The selection of the component types to be included in the *Cubus* is an open question, too. It is done on the basis of the typically data-driven nature of sensor network applications. The component types have to be essential for sensor networks, i. e., they will be used in many scenarios which justifies the inital classification effort. However, it is much easier to add a new component type retroactively than to add or change a dimension of the cubus. Moreover, a single component should provide atomic functionality. Complex functions like aggregation have to be splitted into atomic functions such as the build-up of an aggregation tree, the aggregation control, and the actual aggregation functions.

The biggest effort has to be spent on the classification of the algorithms. We pursue two different approaches: simulation and analysis. Classification by simulation has the advantage that it can be done almost automatically. But due to the number of parameters a complete simulation is not possible – and probably not needed. Strategies have to be developed to reduce the number of needed simulations. In contrast, classification by analysis has to be done manually. Analysis reveals general characteristics of an algorithm, but might be too complex for big algorithms. Moreover, the mostly qualitative results of the analysis are not as easily comparable as the quantitative results of the simulation. For both methods, procedures have to be developed to classify algorithms that depend on other algorithms and change their performance accordingly.

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. In some cases, adaptation can be done locally, while in other cases (neighboring) nodes have to negotiate which introduces additional framework overhead. The synchronized reconfiguration of the adapting nodes is ensured by the *Tiny Configuration Engine*.

4 Related Work

SensorWare [1] and *Impala* [9] are frameworks that 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 and does not support adaptation and cross-layer interaction, 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.

Deluge [7] is a widely-used dissemination protocol that reliably propagates complete code images within a TinyOS-based sensor network. It reduces the number of messages sent by adapting to the density of the network. However, *Deluge* always installs the same code image on all nodes in the network and does not take into account that in real-world applications not all of them have to use the same code.

Reijers and Langendoen [13] describe a scheme to install code on sensor nodes. Their goal is to minimize the size of the code update by transmitting only the differences to the previous version. However, they only consider installing new versions of a complete application rather than replacing the components that constitute it. Therefore, they cannot take some components and install them in another application, for example.

The *MobileMan* project [2] 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* [4] 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.

5 Conclusion and Future Work

In this paper, we have described the architecture and research challenges for the development of TinyCubus, an 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 Cross-Layer Framework*, that provides a generic interface and a repository for the exchange and management of cross-layer information, the *Tiny Configuration Engine*, whose purpose is to manage the upload of code onto the appropriate sensor

nodes, and the *Tiny Data Management Framework*, that provides the required adaptation capabilities.

Furthermore, we have described our role-based code distribution algorithm that uses cross-layer data such as role assignments in order to reduce the number of messages needed to distribute code to certain nodes. Our evaluation shows that the performance of this algorithm is several times better than a flooding approach when the topology and the distribution of roles is well-known.

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, we plan on extending our role-based code distribution algorithm to support highly mobile sensor nodes, such as the ones found in the *Cartalk 2000* project.

References

[1] 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 Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys 2003)*, 2003.

[2] M. Conti, G. Maselli, G. Turi, and S. Giodano. Cross-layering in mobile ad hoc network design. *IEEE Computer*, 37(2):48–51, 2004.

[3] 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*, pp. 1–11, 2003.

[4] 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*, pp. 283–296, 2004.

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

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

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

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. of the 1st Int'l Conf. on Embedded Networked Sensor Systems*, pp. 126–137, 2003.

[9] 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*, pp. 107–118, 2003.

[10] 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 Int'l Conf. on Intelligent Sensors, Sensor Networks & Information Processing*, 2004.

[11] 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*, 2005.

[12] 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, pp. 545–550, 2002.

[13] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. of the 2nd ACM Int'l Conf. on Wireless Sensor Networks and Applications*, pp. 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*, pp. 7–12, 2004.

[15] Sustainable bridges web site. `http://www.sustainable-bridges.net`.

[16] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Comm. of the ACM*, 47(6):34–40, 2004.


1


2


3


4

1 Dr. Pedro José Marrón received his bachelor's and master's degree in computer engineering from the University of Michigan at Ann Arbor in 1996 and 1998, respectively. During his stay in Michigan, he worked as a teaching and research assistant in the areas of databases, compiler construction, and distributed systems. At the end of 1999, he moved to the University of Freiburg in Germany to work on his Ph.D., which he received with honors in 2001. Since 2003, he works at the University of Stuttgart as a senior researcher, where he leads the mobile data management and sensor networks group. His current research interests are distributed systems, mobile data management, location aware computing, and sensor networks. He is a member of the ACM and GI.
Address: Institut für Parallele und Verteilte Systeme, Universitätsstr. 38, 70569 Stuttgart, Germany, Tel.: +49-711-7816-223, Fax: +49-711-7816-424, E-Mail: pedro.marron@informatik.uni-stuttgart.de

2 Dipl.-Inf. Daniel Minder received his diploma in software engineering from the University of Stuttgart in 2003. Since then, he is working as a research assistant in the Distributed Systems group. His Ph.D. research topic deals with efficient data management in sensor networks.
Address: Institut für Parallele und Verteilte Systeme, Universitätsstr. 38, 70569 Stuttgart, Germany, Tel.: +49-711-7816-224,
Fax: +49-711-7816-424,
E-Mail: daniel.minder@informatik.uni-stuttgart.de

3 Dipl.-Inf. Andreas Lachenmann received a master's degree in computer science from the Georgia Institute of Technology, Atlanta, in 2003 and a diploma in software engineering from the University of Stuttgart in 2004. Since then he is a researcher and Ph.D. student at the University of Stuttgart, where he is working on cross-layer architectures for sensor networks.
Address: Institut für Parallele und Verteilte Systeme, Universitätsstr. 38, 70569 Stuttgart, Germany, Tel.: +49-711-7816-280,
Fax: +49-711-7816-424,
E-Mail: andreas.lachenmann@informatik.uni-stuttgart.de

4 Prof. Dr. Kurt Rothermel received his doctoral degree in Computer Science from the University of Stuttgart in 1985. From 1986 to 1987 he spent a sabbatical at the IBM Almaden Research Center working on distributed database management systems. In 1988 he joined IBM's European Networking Center, where he was responsible for several projects in the area of distributed application systems. He left IBM in 1990 to become a Professor of Computer Science back at the University of Stuttgart, where he now leads the Distributed Systems Research Group. Currently, he has published over 110 scientific papers at international conferences and journals. His current research interests are communication architectures and protocols, management of distributed systems, location aware and mobile computing, trust and security. He is a member of the IEEE Computer Society, ACM and GI.
Address: Institut für Parallele und Verteilte Systeme, Universitätsstr. 38, 70569 Stuttgart, Germany, Tel.: +49-711-7816-434,
Fax: +49-711-7816-424,
E-Mail: kurt.rothermel@informatik.uni-stuttgart.de