Life Cycle Support for Sensor Network Applications

Urs Bischoff
Computing Department
Lancaster University
Lancaster, UK
u.bischoff@comp.lancs.ac.uk

Gerd Kortuem
Computing Department
Lancaster University
Lancaster, UK
kortuem@comp.lancs.ac.uk

ABSTRACT

Developing applications for sensor networks is a challenging task. Most programming systems narrowly focus on programming issues while ignoring that programming represents only a tiny fraction of the typical life cycle of an application. Furthermore, application developers face the prospect of investing a lot of time in writing code that has "nothing" to do with the actual application logic. A lot of this code is related to different life cycle concerns such as distributed programming issues or runtime services (e.g. group communication or time synchronisation). In this paper we introduce an engineering method that simplifies the development of sensor network applications by providing comprehensive life cycle support for programming as well as ongoing evolutionary modification of embedded applications throughout the application life cycle. The proposed engineering method is realised in form of a concrete system called RuleCaster. To verify the utility of the engineering method and RuleCaster we use a scenario-based evaluation method.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design

Keywords

Assessment, middleware, sensor networks, separation of concerns, software engineering

1. INTRODUCTION

Wireless sensor networks are one important building block towards the realisation of context-aware and intelligent environments. They bridge the gap between the physical world and the computing system. These networks present the challenge of building large-scale distributed applications tightly-integrated with the physical world.

Research has focused on functionality and operational aspects such as making hardware energy-efficient, developing suitable MAC-protocols and providing generic services such as data aggregation or localisation. Despite impressive research progress it is still very hard to build applications. There are several reasons for this problem.

Programming distributed applications for large-scale networks is difficult. Distributed application code expresses the application logic in terms of a set of distributed tasks for individual nodes. Hence, the application logic has to be decomposed into these tasks. Traditionally, this is done by the application developer. However, manually decomposing an application for thousands or tens of thousands of sensor nodes is not feasible anymore.

Dealing with changing application requirements and a dynamic infrastructure is hard. The application developer is forced to address issues that can be attributed to different life cycles when implementing applications. First, the application logic has to be analysed and separated into a set of distributed tasks for the underlying network. Then, the tasks have to be implemented for the specific hardware which poses a range of runtime problems: common software services and data structures for data aggregation, group communication, time synchronisation, application monitoring etc. have to be coded. Apart from these development time issues, we have to keep in mind the other life cycles of an application such as future changes to the infrastructure and the application requirements.

Middleware approaches provide basic solutions to some of the operational aspects of sensor network applications such as data aggregation or code updates. We argue that these middleware solutions are a good starting point. However, a more comprehensive engineering method is needed that integrates the middleware but also supports the development of large-scale sensor network applications.

The main contribution of this paper is the introduction of a novel engineering method that separates the development of an application into concerns that can be addressed individually. Accordingly, the RuleCaster system — a concrete instance of this process — allows the application programmer to express global application behaviour with global language abstractions. A mapping process analyses the application definition and automatically generates the distributed application code which can be executed by the middleware of the given network infrastructure. The main benefits of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MidSens'07, November 26-30, 2007 Newport Beach, USA
Copyright 2007 ACM 978-1-59593-929-6/07/11 ...$5.00.

this approach are:

- RuleCaster moves away from a node-centric programming model to a network-centric model. Applications are developed for the network as a whole, rather than for each individual node.

- RuleCaster supports different development phases (initial implementation, deployment and maintenance) in a unified way.

This paper is organised as follows. In the following section we propose an engineering method based on separation of concerns. In Section 3 we discuss some related work on programming of wireless sensor networks. Section 4 presents RuleCaster, a system that implements the proposed engineering process. Section 5 addresses the role of the middleware in the proposed engineering method. In Section 6 we first introduce a scenario-based evaluation method and then evaluate RuleCaster and the underlying engineering process. Section 7 concludes this paper.

2. ENGINEERING AN APPLICATION

2.1 Problem Analysis

We envision sensor networks to be an important part of the environment we live and work in. As emphasised by Rodden and Benford [17], living and work environments are subject to continuous transformation: they are modified by the people who inhabit them in a variety of ways, for a variety of purposes and with different frequencies. This observation also has an effect on the requirements of a computing infrastructure. As such, it is subject to the same dynamics as the rest of our living and work environments. As users we want to change or extend the application logic from time to time. With better technology becoming available the network or parts of it should be exchanged without greatly affecting the running applications. Or by adding nodes and replicating tasks we want to make the application more reliable.

Upgrading and modifying embedded software is difficult because of the tight coupling between hardware and software. These application changes take place during three life cycle phases, namely development, deployment and maintenance (cf. Figure 1). Not only do we have to address these change scenarios after the initial deployment, but we also have to consider them when initially developing the application. For example, the application developer has to decide if it is better to implement a centralised or a highly-distributed solution. Or she has to keep future changes to the infrastructure (e.g. addition of nodes) in mind when implementing networking protocols. The former issue can be attributed to the deployment of an application, while the latter one is a maintenance challenge.

2.2 Approach

We argue that these dynamics should be addressed by a suitable engineering method that integrates the different life cycle concerns into a unified system. We have identified three major concerns the above-mentioned application changes can be attributed to, namely logical structure of an application, physical structure of an application and infrastructure (cf. Figure 2).

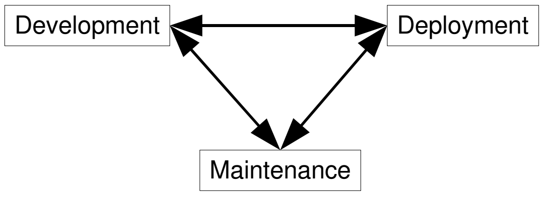

Figure 1: The different life cycle phases of wireless sensor network applications

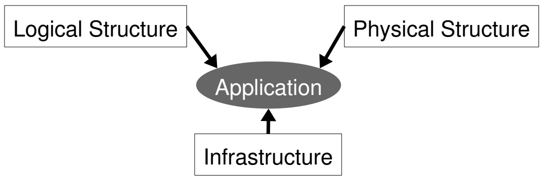

Figure 2: Development concerns

By separating the problem of developing an application for such an environment into these three concerns we can deal with them individually. The logical structure describes how functional elements and application states are connected for describing the application logic. Changes to the application logic refer to changes in the observable behaviour of an application. For example, while an application might initially be defined to open the window when the room is too hot, a new application logic might turn on the air conditioning instead.

The physical structure describes where computational elements are executed and where application states are stored in the network infrastructure. Changes to the physical structure of an application refer to changes in the distribution of computing tasks to individual nodes. For example, a task initially performed by a central node is distributed over several nodes in order to improve reliability and decrease energy consumption.

The infrastructure is the actual sensor network that stores and computes the application states. It is concerned with the runtime of the application whereas the logical structure and the physical structure address development time concerns. Changes to the infrastructure refer to changes in the underlying hardware and runtime system (e.g. network system or middleware). For example, a system might need to be updated when a new generation of hardware devices becomes available with different cpu, memory and radio. Another example is modifying a system by adding or removing nodes.

3. RELATED WORK

One main issue that makes application development for large-scale networks difficult is the decomposition problem. The programmer is faced with distributed programming problems; he has to decide how to express the global behaviour of an application in terms of distributed tasks. This global to local mapping is a delicate problem. Two broad classes of approaches have emerged in the literature to address this problem. The first one focuses on providing high-level abstractions and middleware that make the communication between modules residing on different nodes easier (e.g [19, 15, 18, 14, 8]). The second one consists of a number of *macroprogramming* systems (e.g. [7, 12, 16]). These systems provide abstractions that allow the implementation of an application for the network as a whole, rather than the individual nodes. The middleware approaches focus on infrastructure (cf. Section 2) issues whereas the macroprogramming approaches mainly address the logical structure of an application. Each of these two classes focus on a different concern of our engineering method. We argue that a successful system has to address all three concerns. In Section 4 we briefly describe a concrete system that implements this engineering method. This system is related to these macroprogramming approaches in terms of a language abstraction to describe the logical structure. We integrated it into our engineering method in order to also address the dynamics of the physical structure and the infrastructure.

The need for considering different life cycles in a programming system is addressed in work on distributed software architectures [13]. The main principle is that the configuration language used for structural description should be separate from the programming language used for basic programming. It allows the physical distribution to be specified completely orthogonal to logical structure. This simplifies future changes to components and their distributed structure. Our separation is similar to this. However, we explicitly address dynamic large-scale sensor networks instead of general distributed systems. This allows us to use a high-level language for implementing the application logic for the network as a whole. The physical structure can be automatically derived from this global application definition. This simplifies the development as the physical structure does not have to be explicitly described by the application developer.

4. THE RULECASTER SYSTEM

In this section we briefly describe RuleCaster, an exemplary system that implements the engineering method introduced in Section 2. For more information the reader is referred to [3, 1]. Figure 3 depicts the architecture of this system. The structure of the engineering method is reflected in this architecture.

The RuleCaster Application Language (RCAL) provides language abstractions that address the network as one distributed unit. This allows the application developer to express application logic for the network as a whole instead of the single nodes. Basically the application developer only has to deal with the logical structure of an application. RCAL is based on a state-based programming model [2]. Rules are used to define conditions for transitions from one state to another state. Previous work showed that people are naturally inclined to use rules when asked to describe the behaviour of a smart space [4]. Furthermore, state models are naturally used for describing physical systems [11]. A rule consists of a goal and one or several conditions. Conditions build the elementary building blocks of an application; rules are used to combine these conditions into the logical structure of the application.

As mentioned in Section 2 the logical structure is orthogonal to the physical structure of an application. The logical structure describes global application behaviour. This logical structure can then be transformed into the physical one. Basically we have to decide where in the network application states are computed and stored. Although this

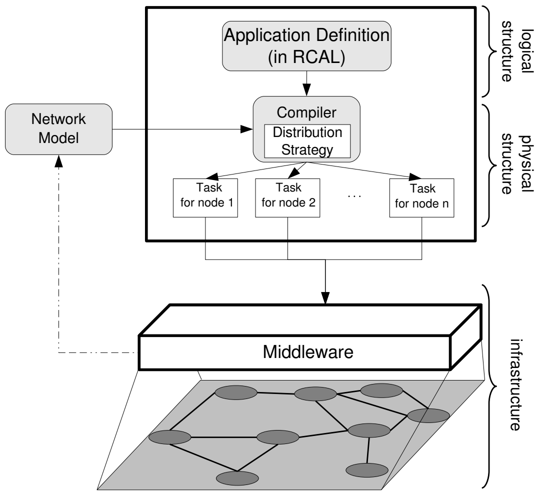

Figure 3: The architecture of the RuleCaster system. The application, which is specified in RCAL is split by the compiler into individual tasks and distributed over the network of sensor nodes. The distribution strategy is a compiler plug-in that influences the distribution process based on some quality model such as energy consumption or robustness.

transformation does not affect the functionality of an application (given the correct implementation), it does affect the quality (or non-functional) aspects of the application such as energy-consumption, reliability, security etc.

This transformation can be done in different ways. The functional elements can be manually assigned to nodes in the network, or an automatic process can find a solution based on a given cost model. We implemented a primarily automatic process influenced by some manually adjusted parameters. This translation is done by the RuleCaster compiler; it transforms the logical structure into a set of tasks. In order to generate and assign the individual tasks, the compiler has to have a view of the network. To achieve this, RuleCaster uses a dynamic network model which provides the compiler with a list of properties (e.g. location, hardware specification etc.) and available services. Services give access to sensors and actuators.

The infrastructure consists of the actual sensor-actuator node hardware running a middleware that executes the application. This middleware is based around a service-based architecture. Services give access to the interface between the network and the physical world (i.e. sensors and actuators).

The method introduced in Section 2.2 is directly reflected in the RuleCaster system. The core of RuleCaster is a high-level compiler system that takes three models as input to generate a distributed software system. Figure 4 depicts these three models. Changes to any of these three models can be directly propagated to the running application by re-compilation and re-deployment.

5. THE ROLE OF MIDDLEWARE

Traditionally middleware is seen as a run-time construct designed to support requirements of distributed applications such as reliable communication, security etc. Because mid-

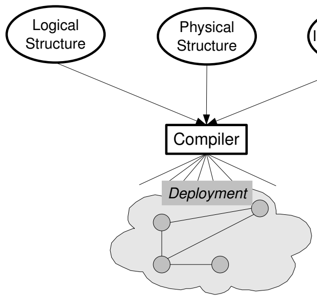

Figure 4: Changes to one of the three models can be directly propagated by re-compilation.

dleware is supposed to support as large as possible set of applications it is often difficult to clearly judge and compare the utility of specific middleware platforms. Different middleware platforms may support different styles of applications and rarely do two middleware platforms target the same application space. The problem of evaluating middleware platforms was highlighted in [6] in the context of infrastructures for interactive applications. The authors argue for the value and use of lightweight application prototypes as a means to more systematically evaluate the utility of middleware platforms.

In the context of RuleCaster, we view middleware from a software engineering perspective, namely as a part of the developers toolset. As such middleware must be designed to support key engineering tasks and integrate with other development tools (in addition to supporting run-time requirements). This provides us with a concrete set of requirements for the middleware and allows us to compare and evaluate individual middleware platforms.

Having a well-defined engineering system in place allows us to clearly specify the set of functions the middleware has to support. Developers using RuleCaster do not program the middleware directly, but they use high-level compiler tools to define applications that are then broken down into individual tasks to be executed by the middleware.

In concrete terms, middleware for the RuleCaster system must support the following functions:

Code distribution. In order to support the automatic deployment of new tasks, the middleware must provide a mechanism for code distribution. RuleCaster's compiler generates individual code for every node. Hence, existing code distribution systems (e.g. Deluge [9]) cannot be used as they send the same code image to every single node. Furthermore, we need mechanisms to upgrade the network with new services; e.g. a high-level audio sensor service, that detects specific noise patterns. Such code should be uploaded to only those nodes that have the required audio sensor and are able to execute the code.

Execution. RuleCaster's compiler generates high-level application code that has to be executed by the middleware. Units of RuleCaster code are called operators. A node has to be able to execute several of these operators (which can belong to the same or different applications) in parallel.

Communication. RuleCaster depends on two high-level communication patterns: *querying* and *informing*. The former one is communication initiated by the sender of information, while the latter one is initiated by the receiver (i.e. query-reply communication). These two forms of communication require peer-to-peer multi-hop networking.

Control access to sensors and actuators. The middleware must provide access to sensors and actuators. The middleware has to transform raw sensor readings into high-level sensor data that can be accessed by the RuleCaster tasks.

Group assignment. A fundamental concept of RuleCaster are groups of nodes (called RuleCaster spaces). Application logic is defined for specific spaces. Nodes are either statically assigned to spaces or the middleware has to dynamically assign nodes to spaces based on some descriptive rules such as 'all nodes in building A that are near a window belong to space XY'.

Monitoring and service discovery. The RuleCaster compiler requires a network model that describes the capabilities and properties of the network. Some of this information is static while other information has to be extracted from the network at runtime. The middleware has to provide a description of services of individual nodes.

Distributed temporal reasoning. A typical sensor network application does some reasoning based on sensor input from several distributed sources. Time intervals are used in RuleCaster to describe relevant relations of distributed sensor events. For example, an application assumes that two detections of a bang by two different nodes are caused by the same source if they are detected within a common interval of 1s.

In order to facilitate an optimal interplay between runtime, development and maintenance aspects of the system we are using our own RuleCaster middleware that implements these very specific functions. It is built on top of the Contiki OS [5]. The core part is a stack-based virtual machine that executes RuleCaster operators.

6. SCENARIO-BASED EVALUATION

In Section 2 we argue that we need an engineering method that addresses the different life cycle concerns. Specific engineering tasks can be attributed to each of these concerns. An evaluation has to focus on how well these tasks are supported. In this section we first describe the evaluation method. It is inspired by work on scenario-based analysis of software architectures [10]. Using this method we then evaluate the utility of our approach by showing how a number of scenarios are supported by RuleCaster and the underlying engineering method.

The fundamental problem is that the quality of a solution is not easily quantifiable. The quality attributes we are interested in are ease of use, long-term maintenance etc. These qualities are very vague and do not provide a clear procedural method to evaluate the engineering process in general and RuleCaster in particular. Furthermore, these quality attributes are context-dependent, meaningful only in the presence of specific circumstances in the life cycle of an application.

Having benchmarks or more universal expressions of quality attributes would be beneficial for an evaluation. Until we reach this understanding we must consider these attributes in the specific contexts the system is used in. To represent contexts, we use scenarios focusing on engineering tasks. Scenarios allow us to express particular instances of a quality attribute important to specific life cycles of an application. The following four steps describe the evaluation method:

1. **Describe scenarios.** A scenario addresses a specific quality attribute that should be evaluated; e.g. a new sensor node is added to the network.
2. **Structure the scenarios.** We determine which concerns each scenario addresses; e.g. the specific scenario addresses the infrastructure of the application.
3. **Perform scenario evaluation.** We show how well the given engineering method and system support the given scenario.

In the following we use the evaluation method to evaluate the utility of our engineering method and system. The following scenarios are taken from a smart home environment. Wireless sensor technology allows us to implement applications such as security, safety or intelligent climate control. To make this discussion more concrete we use the following simple application: *the windows of the building should be opened as soon as the temperature is more than 30 degree Centigrade.*

In this scenario-based evaluation we address the two above-mentioned problems: (1) implementing distributed applications for large networks, and (2) dealing with changing application requirements and infrastructure. We can assume that the infrastructure is installed and the initial application deployed. This allows us to focus on the more interesting change scenarios. Table 1 lists the chosen scenarios and the main concern each of them addresses.

The following list discusses how each individual scenario is supported by our engineering method and RuleCaster. We address each scenario from an application developer point of view:

1. Under the assumption that the outside temperature is available in the network we have to add an additional rule to the application definition in scenario 1. Then, the changed code is recompiled and automatically deployed in the network. We only have to make changes to the logical structure of the application.
2. We do not have to change the application code. The network model reflects the infrastructure changes to the compiler, which can find a better task assignment to the nodes. The changes only affect the physical structure of the application.
3. After having added the new nodes configured with the RuleCaster middleware, we recompile the application and re-deploy it. We do not have to change the application logic because the changes do not affect the logical structure of the application.
4. This scenario requires to change the application logic. The old rules are replaced by the new rules. Then the application is compiled and automatically deployed in the existing infrastructure that supports the execution of the new application logic.
5. The neighbours' network infrastructure is set up differently. They have fewer sensor nodes. Because the application logic stays the same, the application developer only has to re-compile it to generate the adapted physical structure of the application before it is re-deployed.
6. The old sensor nodes are replaced with the new ones. These new nodes are configured with RuleCaster's service-based middleware. Then the application logic is redeployed. The logical and physical structure have not changed. Only the infrastructure is affected by these changes.

7. CONCLUSION

The presented engineering method addresses the initially mentioned problems that make the development of wireless sensor network applications difficult. RuleCaster is a specific system that implements this engineering method. The proposed approach provides comprehensive life cycle support and reduces development efforts through a number of measures:

- By separating the engineering process into well-defined concerns, we can assign specific functions to the individual parts of the system. The middleware in particular has to support a set of functions. This clear separation allows us to deal with the middleware independently of the other parts; i.e. we can also address non-functional aspects such as performance or reliability without affecting the functionality of the applications.
- By separating the logical structure from the physical structure we can implement the application logic without being directly faced with distributed programming problems. RuleCaster's high-level language RCAL hides the application definition from specifics related to the infrastructure. These two parts can be implemented independently. Hence, the application developer can focus on the actual application logic, which simplifies application development.
- RuleCaster provides network-level language abstractions. This allows the programmer to directly implement global application logic without being forced to express it in terms of distributed tasks.
- RuleCaster automates the deployment process: changes to any of the three models (logical structure, physical structure, infrastructure) can be directly propagated to the running application by re-compilation and re-deployment. This simplifies ongoing evolutionary modification of applications throughout the application life cycles.

Research into wireless sensor networks has made impressive progress in terms of functionality. There are several middleware solutions that provide support for runtime issues such as data aggregation, localisation etc. We also have to address programmability and maintainability if we want to make this technology successful. The proposed solution integrates these aspects into one unified engineering method.

Table 1: Change scenarios.

	Change scenario	Example	Concern
1.	Extend application.	Only open windows if outside temperature is less.	Logical structure
2.	Change the distribution of the application.	Some nodes have been connected to a permanent power supply. They should therefore contribute more computational power to the execution of the application.	Physical structure
3.	Add new nodes to the infrastructure.	New temperature sensor nodes are added to have finer-grained temperature measurements.	Infrastructure
4.	Exchange application.	The new inhabitants of the house do not like the application. They only want to use the available humidity sensors to decide when to turn on a dehumidifier.	Logical structure
5.	Move application.	The neighbours are interested in running the same application on their wireless sensor network infrastructure.	Physical structure
6.	Exchange parts of the network.	New energy-efficient humidity sensor nodes are available. The old humidity sensor nodes should be exchanged with the new ones.	Infrastructure

8. REFERENCES

[1] Urs Bischoff and Gerd Kortuem. A compiler for the smart space. In *Proceedings of AmI-07*, 2007.

[2] Urs Bischoff and Gerd Kortuem. A state-based programming model and system for wireless sensor networks. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, 2007.

[3] Urs Bischoff, Vasughi Sundramoorthy, and Gerd Kortuem. Programming the smart home. In *Proceedings of the 3rd IET International Conference on Intelligent Environments*, September 2007.

[4] Anind D. Dey, Timothy Sohn, Sara Streng, and Justin Kodama. iCAP: Interactive prototyping of context-aware applications. In *Proceedings of the 4th International Conference on Pervasive Computing (PERVASIVE 2006)*, 2006.

[5] Adam Dunkels, Björn Grönwall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, 2004.

[6] W. Keith Edwards, Victoria Bellotti, Anind K. Dey, and Mark W. Newman. The challenges of user-centered design and evaluation for infrastructure. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 297–304, New York, NY, USA, 2003. ACM Press.

[7] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using Kairos. In *Distributed Computing in Sensor Systems: First IEEE International Conference (DCOSS 2005)*, 2005.

[8] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network Magazine Special Issue*, 18:6–14, 2004.

[9] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of SenSys'04*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[10] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, November 1996.

[11] Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. State-centric programming for sensor-actuator network systems. *Pervasive computing*, pages 50–62, 2003.

[12] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[13] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[14] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks*, 2005.

[15] Luca Mottola and Gian Pietro Picco. Logical neighboorhoods: A programming abstraction for wireless sensor networks. In *DCOSS*, 2006.

[16] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[17] Tom Rodden and Steve Benford. The evolution of buildings and implications for the design of ubiquitous domestic environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 9–16, New York, NY, USA, 2003. ACM Press.

[18] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[19] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, 2004.