A State-based Programming Model and System for Wireless Sensor Networks

Urs Bischoff, Gerd Kortuem
Computing Department
Lancaster University
{u.bischoff, kortuem}@comp.lancs.ac.uk

Abstract

Sensor networks are one important building block towards the realisation of context-aware applications. Suitable communication protocols and middleware solutions are necessary to facilitate the development of sensor network applications. Additionally, the vast number of nodes in sensor networks necessitates a new programming model for application developers. Commonly used programming abstractions force the programmer to express the global behaviour of a network in terms of local actions taken at individual nodes. We argue that global programming abstractions are needed to express global behaviour of a network. We present RuleCaster, a novel state-based programming framework for sensor network applications. RuleCaster provides a high-level language for application definition, a compiler that splits the high-level specification into individual tasks and assigns them to the nodes, and a service-based middleware that provides the interface for collaborative execution of these tasks.

1. Introduction

Context-awareness is an important concept in pervasive computing. Sensed information from our environment can be used to adapt the behaviour of an application. Sensor networks are one building block towards the realisation of pervasive computing applications [13]. Research into sensor networks has made impressive progress in several areas, including small low-powered hardware design (e.g. [9, 3]), wireless networking (e.g. [19, 15]) and software services (e.g. [14, 12]). These advancements allow us to move beyond centralised processing of aggregated sensor data to distributed processing. The execution of application logic can be "pushed" into the actual network, closer to where sensor data is collected. The idea of ubiquitous sensor systems raises one key research question: *how can we write, deploy and maintain an application for thousands or tens of thousands of sensor nodes efficiently?* It is no longer practical to program and re-program each node individually. This would be too time consuming, costly and error prone, especially in a dynamic environment with constantly changing application configurations.

Macroprogramming [11] has emerged as one promising approach. It describes a method for specifying the global behaviour of an application by providing network level programming abstractions. Instead of focusing on each individual node independently, the network is programmed as one unit. We see a state-based programming model as a useful abstraction for macroprogramming. Liu et al. argue that the notion of states of physical phenomena and models of their evolution over space and time are deeply rooted in typical wireless sensor network applications [5]. They describe a system for managing state variables in a distributed sensor network. We use the notion of states and state transitions in a macroprogramming language abstraction. Observations at nodes determine the state of the network. Rules are used to define network state transitions. The advantage of this high-level approach is that distribution aspects and the decision where states are stored and manipulated are hidden from the application developer, who can therefore focus on the implementation of the actual application logic.

In this paper, we introduce the design and implementation of *RuleCaster*, a novel macroprogramming framework for wireless sensor networks. Applications are defined for the network as a whole. A compiler then splits the application definition into individual tasks that form the executable application code. The novelty of our approach is that the developer specifies the application logic without having to decide where tasks are executed in the network and how nodes communicate. This decision is left to the compiler that generates the tasks based on a chosen distribution strategy so that applications can exhibit specific properties in terms of robustness and efficiency.

The remainder of the paper is as follows. Section 2 introduces the state model that builds the basis of our system. In Section 3 we describe the system. It is based on a high-level language, a compiler and a middleware that provides the interface to execute the compiled code. Section 4 shows how

the compiler has the freedom to decide where state variables are stored and manipulated inside the network based on specific application requirements. Advantages and problems of our approach are addressed in Section 5. Section 6 compares our approach to related work and Section 7 concludes this short paper.

2. State Model

Our state-based programming system was inspired by research into multiagent systems and distributed AI [16]. A network consists of nodes which have sensors to observe and actuators to affect the environment. It is partitioned into several spatial regions. Each region is in one or several discrete states. The union of all these states at time t defines the network state at time t. Rules are used to specify the conditions for a state transition. More formally we can denote a basic state of region r at time t as $s_{t,r} \in S$, where S is a discrete set of possible states. The union of all basic states of region r at time t describes the region state $S_{t,r}$. The network state S_t at time t is the union of all region states $S_{t,r}$ at time t. Changes in the environment can cause the network to change state: $S_t \rightarrow S_{t+1}$. These transitions are specified as a set of rules. A rule describes the transition of a basic state:

$$s_{t,r'} \rightarrow s_{t+1,r} \tag{1}$$

r and r' can be the same or different spatial regions. A rule is a boolean function consisting of disjunctions and conjuntions of boolean conditions $c_{i,t_i,r}$, where i is the ID of the condition, t_i the time when the condition evaluates to true (or false) and r the region where the condition evaluates to true (or false). The outcome of a condition evaluation at time t depends on an observation o_t at time t (e.g. a sensor reading). Time is essential because the outcome of a condition evaluation is time-dependent. Therefore the successful evaluation of a specific rule is restricted to a pre-defined time interval $\triangle t$, where $t_1, t_2, \ldots, t_n \in [t - \triangle t, t]$.

(1) shows the transition from one basic state to another basic state. A rule can also describe the more general case of an n-to-1 transition, the transition from a set of basic states $S'_{t,r'} \subset S_{t,r'}$ to one basic state $s_{t+1,r}$. If $S'_{t,r'} = \{\}$, the transition rule specifies the generation of a new basic state.

States are generally partially observable. This means that nodes have to collaborate to determine the state, which is done through collaborative evaluation of transition rules.

An action $a_{t+1,r}$ is triggered by a valid state $s_{t,r'}$. An action can indirectly change the network state through affecting the environment.

3. The State-Based Programming System

Based on the model introduced in Section 2 we developed the RuleCaster system. A high-level language allows the implementation of state-based applications; rules are used to specify state transitions. A service-based middleware provides the interface to execute the compiled application code. RuleCaster treats the network as one distributed computer. The advantage of this approach is that the programmer does not have to specify where states are stored or manipulated. This decision is made by a compiler that generates the executable application code. Figure 1 gives an overview of the architecture.

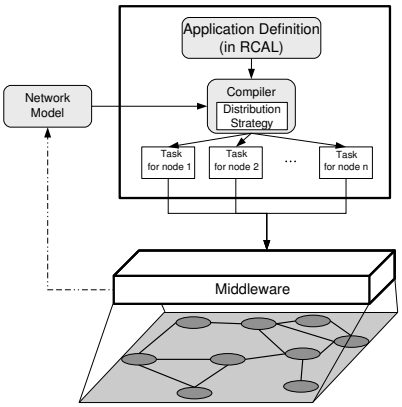

Figure 1. The architecture of the RuleCaster system. The application, which is specified in RCAL is split by the compiler into individual tasks and distributed over the network of sensor nodes. Distribution strategies can vary based on the application requirement, such as minimum energy consumption, or the level of robustness.

In RuleCaster, applications are defined in RCAL (RuleCaster Application Language). Before the application can be executed by the network, the compiler decides where states are stored and manipulated and accordingly splits the application into several tasks. Each task is individually assigned to a selected node, with a specified role in the execution of the whole application. They define where network states are stored and how transition rules are evaluated. Finally, a service-based middleware provides the interface and runtime environment to execute these 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.

We now describe in further detail, the four parts of RuleCaster: (1) the high-level language, RCAL, (2) the compiler, (3) the network model and (4) the middleware.

3.1. Application Language

RCAL is a high-level language for specifying state transitions. Applications are defined through a set of state transitions and actions. However, RCAL applications do not specify where and how states are manipulated and stored in the network. An RCAL application definition consists of several *ruleblocks*. Each ruleblock specifies a state transition or an action.

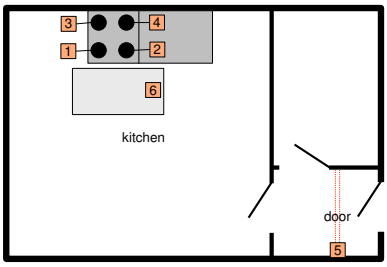

Figure 2. The layout of a simple sensor network.

To clarify RCAL we use an example of a home application with 6 sensor nodes. The goal of the application is to activate the alarm when the sensor nodes detect that the kitchen stove is switched on, nobody is monitoring the stove, and someone is leaving the building. Figure 2 shows the layout of the example sensor network. The nodes are grouped into two spaces: *kitchen* and *door*. Nodes 1 to 4 detect whether the stove is switched on. Node 6 is a pressure mat on the floor that can detect the presence of an object on top of it. Node 5 is connected to two break-beams; it can determine whether someone is entering or leaving the house.

Example 1 shows three ruleblocks. The first two specify state transitions, the last one an action. In the first ruleblock, *SPACE(kitchen)* defines the spatial region r. *TIME(1s)* sets the time interval $\triangle t$ to 1s; this means that all conditions have to be satisfied within an interval of 1 second. The actual rule is specified inside the ruleblock. *STATE stoveOnHazard* is the target state $s_{t+1,kitchen}$ of the transition. There is no source state $s_{t,r'}$. Thus, the rule specifies the generation of a new state. *stoveOn()* and *notMonitored()* are the conditions $c_{1,t_1,kitchen}$ and $c_{2,t_2,kitchen}$ that have to be satisfied in order for the transition to take place, or in this case to create a new state *stoveOnHazard*. *STATE(kitchen:stoveOnHazard)* in ruleblock 2 defines the source state $s_{t,r'} = s_{t,kitchen} = stoveOnHazard$ at time t of the transition rule. If the condition *leaving()* is satisfied, there is a transition to *STATE hazard* ($= s_{t+1,r}$, where $r = door$ is defined by *SPACE(door)*). Ruleblock 3 defines an action: if the state *hazard* $\in S_{t,door}$, then the action *alarm(10)* should be triggered in the spatial region *door* at $t + 1$.

```
/* Ruleblock 1*/
SPACE(kitchen),TIME(1s){
  STATE stoveOnHazard :- stoveOn(),
                         notMonitored().
  notMonitored() :- pressure(X), X<50.
}

/* Ruleblock 2*/
SPACE(door),STATE(kitchen:stoveOnHazard){
  STATE hazard :- leaving(). }

/* Ruleblock 3*/
SPACE(door),STATE(door:hazard) {
  ACTION alarm(10). }
```

Example 1: An example of an application definition. The objective of this application is to actuate the alarm, when someone is leaving the house (*leaving()*), the stove is switched on (*stoveOn()*), and the pressure on the floor mat is less than 50. (*notMonitored()*).

3.2. Network Model

The network model is used by the compiler (cf. Section 3.3) to make decisions where network states are managed within the network. Based on the individual properties of nodes described in the network model, the compiler generates the tasks and assigns them to the corresponding nodes.

The network model is the interface of the network. It specifies available services, node properties (location information, hardware specification etc) and network properties (connectivity, communication protocols etc). The services are used to evaluate the transition conditions. We assume that the network model is generated by a self-monitoring network infrastructure. Each node in the network provides a description of its properties and services. Other sources of information are used to complete the network model; known location of a node is an example of an alternative source. Changing parts (e.g. energy level of nodes) of this model are dynamically updated while static parts (e.g. hardware specification) are kept the same during the lifetime of the network.

3.3. Compiler

The RuleCaster compiler translates an RCAL application definition into code to be executed by the nodes in the

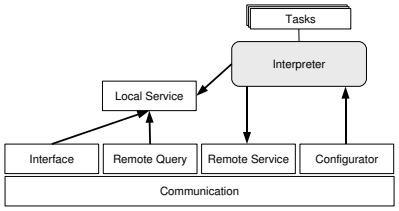

Figure 3. The architecture of the runtime system.

network (i.e. executable application code). Executable application code is represented as a set of tasks; a task is generated for an individual node. A task specifies how a transition rule is evaluated and where states are stored. As seen from Example 1, the application programmer does not have to specify where a network state is manipulated and stored. The spatial region r only defines where the conditions and states are valid; however, rule evaluation does not have to be done in the region itself. Therefore, it is possible to develop several task distribution strategies, based on the application requirements. An example of a requirement is minimal energy consumption. Our goal is to allow the compiler to dynamically assign tasks based on the requirements of the application. This separation of different concerns makes RuleCaster more flexible with respect to changing application requirements compared to more traditional ways of writing software where the distribution of state variables is inherently contained in the application definition. If the requirement is energy efficiency, the compiler finds a solution that minimises the energy consumption of the running application. This is a non-trivial job for the compiler, as there are several dependencies for producing the required result; the type of network (e.g. broadcast vs. unicast protocols), processing cost, communication cost, the different levels of energy in existing nodes, etc. Trade-offs also need to be considered, such as whether to reduce robustness because redundant processing increases energy consumption.

3.4. Middleware

The middleware provides the interface to the service-based architecture of the system. It consists of collaborating runtime systems being executed on the nodes. Tasks are sent to the nodes in a binary representation. The runtime system receives the assigned tasks and starts the execution. Furthermore, the middleware is responsible for describing the nodes' properties and services, which can be used to generate the network model.

Figure 3 illustrates the modular architecture of the runtime system that is executed on the nodes. The core component is the *Interpreter* module. It is responsible for the interpretation of the tasks. It uses the *Local Service* if a condition of a transition rule has to be evaluated locally. The *Remote Service* module queries services running on other nodes if the tasks specifies that a transition condition has to be evaluated by another node. The runtime system also accepts service requests from other nodes through the *RemoteQuery* module. The *Remote Query* module collaborates with the *Local Service* module for this purpose. New tasks generated by the compiler are sent to the *Configurator* module. It decodes the tasks and forwards them to the *Interpreter* module. Finally, the *Interface* module announces properties and available services to the discovery mechanism used to generate the network model.

4. Distribution Strategies

In Section 3.3 we explained that the RuleCaster compiler can generate different executable application codes from the same RCAL application definition. The application programmer can choose a specific compiler plug-in that influences this distribution. We introduce two distribution strategies to illustrate how different executable application codes can be formed from a single application definition. These are (1) Centralised Distribution Strategy, and (2) Decentralised Distribution Strategy. The first strategy uses *service availability* as a factor to generate the executable application code. The second strategy also considers *location* as a parameter. The centralised distribution strategy creates an overlay network with a star-topology: one central node is responsible for all rule processing while the other nodes only provide their services to the central node for condition evaluation. The decentralised distribution strategy generates an overlay network consisting of several sub-networks. Processing is distributed over several nodes that are responsible for their own spatial region.

The Centralised Distribution Strategy increases communication cost, but reduces overall processing cost. On the other hand, the Decentralised Distribution Strategy reduces communication cost, but has redundant processing. Distributed evaluation of rules in the area where relevant data is gathered can be seen as aggregation of high-level semantic data. In-network aggregation of data has been shown to improve the robustness of an application [6].

As mentioned earlier, energy is an important efficiency criterion in wireless sensor networks. In the current version of our RuleCaster compiler, we do not use an explicit energy cost model to produce the optimal solution. However by using the location parameter, we can produce a useful heuristic to decrease energy consumption. Ahn et al [1] show that processing events (such as evaluation of rules) where they occur in the network outperforms a centralised solution in terms of energy.

5. Discussion

A high-level programming language like RCAL can simplify programming of applications. However, there is a tradeoff between the level of abstraction and the expressiveness of a language. A higher degree of abstractions generally means less influence of the programmer on the actual executable application code. We believe that a state-based abstraction addresses a broad class of wireless sensor network applications. Rules have been shown to be intuitive; Dey et al. even argue that people are naturally inclined to use rules when asked to describe the behaviour of a smart space [2].

The advantage of a declarative language like RCAL is that the programmer does not have to specify how the application is executed. The implementation specifies the overall goal of the application, or what has to be executed by the network. An argument against the use of declarative languages is the lack of good debugging tools or exception handling mechanisms.

Adaptation of our middleware is rather limited to a few reactive mechanisms like choosing the next collaboration peer if the first one is not available. These alternatives are determined at compile time. Improved adaptation would be beneficial for the stability of the system. Adaptation of our tasks is a complex problem that goes beyond changing the routing path or the set of actively sensing nodes. The essential problem is that we have to give the nodes enough knowledge so that they can decide how much they are allowed to change the tasks without changing the overall goal of the application. The advantage of RuleCaster is that major changes of the infrastructure can be accomodated by recompilation of the RCAL application without changing the actual application definition.

We showed that the RuleCaster compiler can generate different executable application codes from the same application definition based on application requirements. We used two factors to influence the compilation process: service availability and location. We are investigating the use of additional factors to influence the generation of the executable application code. The question remains how we can measure the quality of a solution with respect to requirements of an application.

6. Related Work

Related work can be subdivided into two main classes: programming support and runtime support. The former class is concerned with providing the programmer with programming abstractions of the nodes and the network. The latter class is focused on providing runtime tools that simplify the deployment and the execution of applications. Both of these classes are interrelated. Middleware solutions traditionally focus on problems of the latter class. There are several middleware solutions that provide services for sharing data within a group of nodes [18, 10, 17]. TinyCubus implements methods for runtime adaptaption of applications [8]. Furthermore, it provides tools for efficient code dissemination in a large sensor network. MiLAN is another middleware example, which proactively adapts the execution roles of nodes to the changing environment [4]. These middleware solution provide useful methods for data sharing and efficient execution of applications. However, they do not absolve the programmer from making decisions where and how the application is executed in the network. The programmer has to decide at implementation time where state variables are manipulated and stored. With a large number of nodes and different roles it makes it hard to update or change an application. Suitable programming support is needed to overcome these shortcomings.

An examplary approach that combines concerns of both classes is TinyDB [7], an SQL-like language abstraction. The sensor network is seen as a distributed database. SQL-like query statements can be used to collect sensor data from the network. While this system provides useful tools for collecting sensor readings, it does not focus on general purpose application execution in the network. Similar to TinyDB, RuleCaster also addresses programming support as well as runtime support. On the one hand, the service based architecture of the middleware implements tools for rule evaluation and code dissemination. On the other hand, RCAL and the RuleCaster compiler provide methods for implementing applications and the generation of efficient executable application code. Additionally, RuleCaster supports sensing, evaluation as well as actuation inside the network.

In terms of the declarative nature of the language, RuleCaster is similar to Regiment [11]. Regiment is based on a functional language. Similar to RuleCaster, Regiment allows the programmer to define the application logic for the network as one unit. The authors argue that functional languages are intrinsically more compatible with distributed implementation over volatile networks than imperative languages. Regiment provides node-dependent operations, which makes it less flexible compared to RuleCaster for distributing tasks in the network. Furthermore, the programmer is required to define operations on raw sensor data in order to interpret their meaning. This makes it difficult to quickly change these operations. Our service-based approach absolves the programmer from making these low-level decisions when implementing an application; the programmer deals with data on a semantic level. Nodes could even provide the same service using two different low-level methods. This gives the RuleCaster compiler more freedom in assigning tasks to nodes.

7. Conclusion

Commonly used programming support abstractions force the programmer to express the global behaviour of a network in terms of local actions taken at individual nodes. This method is cumbersome because the programmer has to deal with many issues related to distributed programming which make application development difficult and error-prone. Furthermore, the lack of standard programming abstractions require the programmer to have an intimate knowledge of the specific hardware platform. To simplify application development for sensor networks, RuleCaster does not focus on individual nodes; instead, the network is programmed as a single unit.

In this paper, we present RuleCaster, a rule-based programming framework for sensor network applications. RuleCaster (1) abstracts the network as a single distributed entity, (2) separates application logic and distribution logic, and (3) easily accommodates changes to an existing application through automatic propagation of changes made to one global application definition. RuleCaster allows the application to be programmed for the network, instead of individual nodes. Therefore the programmer is absolved from making decisions on distributed programming issues (such as data synchronisation, data consistency, etc). The RuleCaster compiler takes an application definition written in RCAL, splits it into individual tasks, and delivers them to the nodes in the network. Tasks are generated by using a specific distribution strategy. A distribution strategy describes an algorithm to find an optimal distribution of tasks based on specific input parameters (such as location). As a result, our method leads to a separation of concerns; application logic and distribution logic can be changed independently of each other. Therefore, RuleCaster easily propagates changes in the application definition to the executable application code through recompilation. Similarly if requirements change, a distribution strategy that caters for the new requirements can be chosen. We believe that RuleCaster is a step towards simplifying the design, development and maintenance of sensor network applications.

References

[1] S. Ahn and D. Kim. Proactive context-aware sensor networks. In *Proceedings of EWSN'06*, 2006.

[2] A. D. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. In *Proceedings of PERVASIVE'06*, 2006.

[3] ETH Zurich. Btnodes - a distributed environment for prototyping ad hoc networks. http://btnode.ethz.ch, September 2005.

[4] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network Magazine Special Issue*, 18:6–14, 2004.

[5] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive computing*, pages 50–62, 2003.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Wong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. In *Proceedings of OSDI'02*, 2002.

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

[8] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of EWSN'05*, 2005.

[9] moteiv. tmote sky. http://www.moteiv.com, September 2005.

[10] L. Mottola and G. P. Picco. Logical neighboorhoods: A programming abstraction for wireless sensor networks. In *Proceedings of DCOSS'06*, 2006.

[11] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceedings of DMSN'04*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[12] K. Römer. Time synchronization in ad hoc networks. In *Proceedings of MobiHoc'01*, pages 173–182, New York, NY, USA, 2001. ACM Press.

[13] G. Roussos and M. Zoumboulakis. Ubiquitous computing and databases: Critical issues and challenges. *Encyclopaedia of Databases*, 2004.

[14] Y. Shang and W. Ruml. Improved MDS-based localization. In *Proceedings of INFOCOM'04*, March 2004.

[15] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of SenSys'03*, pages 171–180, New York, NY, USA, 2003. ACM Press.

[16] N. Vlassis. A concise introduction to multiagent systems and distributed ai. Technical report, Intelligent Autonomous Systems, Informatics Institute, University of Amsterdam, 2003.

[17] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of NSDI'04*, 2004.

[18] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of MobiSys'04*, pages 99–110, New York, NY, USA, 2004. ACM Press.

[19] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of INFOCOM'02*, 2002.