RuleCaster: A Macroprogramming System for Sensor Networks

Urs Bischoff
Lancaster University
Lancaster
LA1 4YW
u.bischoff@comp.lancs.ac.uk

Gerd Kortuem
Lancaster University
Lancaster
LA1 4YW
kortuem@comp.lancs.ac.uk

ABSTRACT

Progress in the development of small low-powered hardware, wireless networking, sensor technology and software services make it possible to build radically new kinds of applications. These sensor network applications are built on top of a large number of low-powered sensor nodes. Writing software for this kind of network can be a challenging task. 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 propose RuleCaster — a system for programming sensor network applications. Instead of writing applications for the networked node, RuleCaster allows to implement an application for the network as a whole. A compiler automatically splits this application specification into several individual tasks and distributes them in the network. Nodes execute the individually assigned tasks.

1. INTRODUCTION

A lot of progress has been made in sensor network research in recent years. Advances in the development of small low-powered hardware, wireless networking and software services (e.g. localisation, time synchronisation) allow us to move beyond data-driven centralised sensor network solutions. The execution of application logic can be "pushed" into the actual network — closer to where actual sensor data is collected. The network is not just a simple tool for data collection; the network is an intelligent tool for data processing. Despite all this research progress it is still difficult to write and easily deploy applications. Sensor networks consist of small, low-powered nodes each with limited computation, sensing and communication abilities. From a programmer's point of view we face one important problem: how can we implement an application for a large number of these sensor nodes? What we need are suitable methods for designing, implementing and deploying applications.

Programming each individual node is not feasible. This would be too time consuming, too costly and too error prone. This is even more relevant if we consider a dynamic environment with often changing application configurations.

There are no well-established protocols and services. It is difficult to define protocols because different applications can have totally different requirements. Existing networking protocols (e.g. TCP/IP) were not designed for this kind of network. These are reasons why a lot of applications have to be built from scratch.

Submitted to OOPSLA 2006 Workshop on Building Software for Sensor Networks.

Sensor network applications are generally implemented bottom-up; applications are built for the single sensor node that uses the network for communication. This does not reflect the way a sensor network application is designed. We argue that instead of focusing on each individual node independently we need to focus on programming the entire network as a whole. Knowing the individual node that executes the application is of less interest.

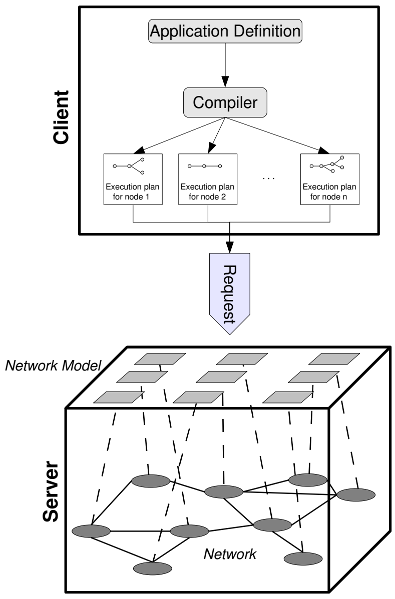
We propose a solution that addresses the network as a whole. Rather than expressing global behaviour of the sensor network in terms of complex, local node-level actions, we envision a global approach. This approach is based on four concepts: (1) a high-level language for application definition, (2) a dynamic model of the sensor network describing its nodes' capabilities and location, (3) a compiler that splits the application into individual tasks and assigns them to nodes and (4) a run-time system that allows the nodes to receive new tasks and execute them in collaboration with other nodes in the sensor network.

The key advantage of our approach is threefold. Firstly, the high-level abstraction reduces the mental gap between the design model of the application and the application language. Secondly, non-relevant hardware details are hidden from the application programmer; hardware specific optimisations, for example, are automatically dealt with by our system. And finally, by being able to focus on programming the network as a whole, the programmer does not have to deal with low-level details (e.g. synchronisation, data consistency, different hardware platforms etc.) which make the development of distributed applications difficult and error-prone.

2. THE RULECASTER SYSTEM

We have developed RuleCaster, a system for implementing applications for sensor networks. RuleCaster deals with the network as one distributed entity. The whole network can be seen as a server that can execute requests from clients to run applications. Basically the server is a highly distributed computer that has to be programmed. Internally this network is based on a service-oriented architecture. A service is a component that simply transforms input events into output events, adding new semantic information as necessary; they can transform low-level sensor events into semantically meaningful high-level events. Figure 1 depicts the architecture of the RuleCaster system.

In the following sections the four core parts — high-level



Client
Application Definition
Compiler
Execution plan for node 1
Execution plan for node 2
...
Execution plan for node n
Request
Network Model
Server
Network

Figure 1: The architecture of the RuleCaster system.

language, compiler, network model and runtime system — of RuleCaster are described in more detail.

2.1 RuleCaster Application Language

RCAL (<u>R</u>ule<u>C</u>aster <u>A</u>pplication <u>L</u>anguage) is a high-level language used to implement sensor network applications. Its unique feature is that it does not specify where and how the application is executed in the network; it only specifies what the network as a whole has to do and what results the client expects. An application is expressed in terms of rules. Rules are natural — they can be understood by both humans and computers [1].

Example 1 An example of a simple application definition.

```
SPACE(kitchen), TIME(SIMULTANEOUS),
  STATE(kitchen:normal) {
    STATE stoveOn :- stoveOn(), monitored().
    monitored() :- pressure(X), X>50.
}

SPACE(door), STATE(kitchen:stoveOn) {
    STATE hazard :- leaving().
}

SPACE(door), STATE(door:hazard) {
    ACTION alarm(10).
}
```

Example 1 illustrates an application for a sensor network in a home environment. It detects when there is a safety hazard because the stove is on and nobody is monitoring it; it then triggers an alarm when a person is leaving the house.

An RCAL application consists of several rule blocks (there are three rule blocks in Ex. 1). Each rule block can specify a space, time and state constraint. These constraints have to be satisfied in order to evaluate the rules inside the rule block. *STATE(door)*, for example, constrains the evaluation of the rule to a space called *door*. *TIME(SIMULTANEOUS)* specifies that all conditions of a rule inside the rule block have to be satisfied simultaneously. The state constraint is a pre-condition that has to be satisfied; in the given example one of the network states has to be *door:hazard* in order to trigger an action (i.e. *ACTION alarm()*). A rule consists of a goal and conditions. The goal is reached if all conditions are satisfied. A special type of rule is the state rule (labelled with the STATE keyword). If a state rule is satisfied, the state is added to the current network states.

2.2 Network Model

The second core part of our approach is the network model (cf. Fig. 1). It is the interface of the network. The model specifies available services and properties of the network. We assume that the network model is generated and dynamically updated 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 (e.g. known location of a node).

Example 2 Description of node 6 in the network model.

```
STATIC PROPERTY(ID, 6);
STATIC PROPERTY(TYPE, "tmoteSky");
DYNAMIC PROPERTY(CONNECTIVITY, 5);
DYNAMIC PROPERTY(ENERGY, 5);
DYNAMIC SERVICE("leaving");
DYNAMIC SERVICE("entering");
DYNAMIC SERVICE("alarm", IN);
STATIC SPACE("door");
...
```

Example 2 shows the interface of one node in the network model. Node 6, for example, can determine whether someone is leaving (*DYNAMIC SERVICE("leaving")*) or entering (*DYNAMIC SERVICE("entering")*) the house. The keyword *DYNAMIC* means that a call to this service can have a different effect at different points in time. Actuators are also announced as services: e.g. *DYNAMIC SERVICE("alarm", IN)*. It accepts one input parameter; here, it is the sound volume of the alarm.

2.3 RuleCaster Compiler

The third core part of our approach is a process that dynamically generates the distributed application. It splits an RCAL-application into several tasks. Analogously to a compiler for a single device application the compiler translates the rule-based application into a distributed application for the network given by the network model.

The compiler generates an individual task for each node. A task is represented as a set of execution plans. An execution plan is a model of how a device has to evaluate a rule that defines a network state.

Figure 2 shows the layout of our example sensor network. It is a small network consisting of 6 nodes. Nodes 1-4 can detect whether the stove is turned on. Node 6 is a pressure mat on the floor that can detect if something is on top of it. Node 5 is connected to two break-beams; it can determine whether someone is entering or leaving the house.

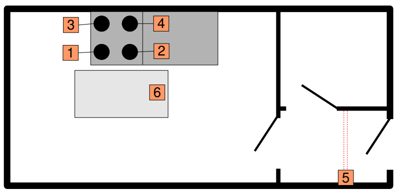


Figure 2: The layout of a simple sensor network.

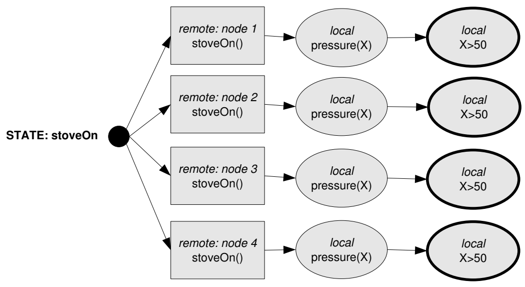


Figure 3: An execution plan in the task sent to node 6.

The RCAL-rule does not specify where the tasks are evaluated. Several distribution strategies are possible: a centralised or a distributed solution are two examples. In the centralised solution, the compiler generates only one task: a task for node 6. In order to evaluate certain conditions node 6 has to query the other nodes. The second strategy is distributed. The nodes in the kitchen evaluate the rules that are concerned with the detection of the kitchen hazard; nodes 1-4 have to collaborate with node 6 in order to detect such a situation. The task of node 5 detects when someone is leaving the building. It then queries one of the kitchen nodes whether there is a potential hazard in the kitchen. If the reply is positive, it triggers an alarm. The centralised solution requires more communication but less processing on a single node. The distributed solution reduces the communication cost; however, redundant processing is possible. The advantage of the distributed solution is its robustness.

Figure 3 illustrates an example of an execution plan in the task of node 6. This directed graph represents the state rule *stoveOn*. The directed edges mark the dependencies of the conditions, which are shown as tree nodes in this representation. If one leaf condition (i.e. no other condition depends on this condition) is satisfied the whole state rule is satisfied. Some of the conditions (e.g. *stoveOn()*) in the execution plan cannot be evaluated locally. Thus, node 6 has to query services running on nodes 1-4.

Many more distribution strategies are possible. An optimal distribution depends on the application requirements: minimal energy consumption, minimum number of messages or reliability are examples.

2.4 RuleCaster Runtime System

The RuleCaster runtime system is running on each node in the sensor network. It is responsible for executing the assigned tasks. Tasks are sent to the nodes by the client's compiler in a binary representation — a serialised form of the execution plan tree. The runtime system receives the assigned tasks and starts the execution. Furthermore it can describe its properties and services, which is used for the generation of the network model.

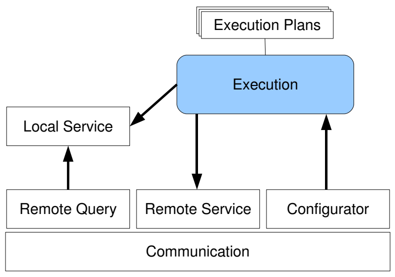


Figure 4: The architecture of the runtime system.

Figure 4 illustrates the modular architecture of the runtime system. The *Execution* module is responsible for the execution of the execution plans. It uses the *Local Service* if a condition can be evaluated locally. If the execution plan specifies a remote condition, it uses the *Remote Service* module to query a service running on other nodes. The runtime system also accepts service requests from other nodes; the *Remote Query* module in collaboration with the *Local Service* module deals with these requests. Finally the *Configurator* accepts new tasks generated by a compiler. It decodes the tasks and forwards the contained execution plans to the *Execution* module.

Example 3 Example of an execution plan representation received by node 6 (cf. Fig. 3).

```
0:  stoveOn() @ node 1
1:  branch 6
2:  pressure(X) @ local
3:  branch 6
4:  X>50
5:  END
6:  stoveOn() @ node 2
7:  branch 12
8:  pressure(X) @ local
      ...
```

Example 3 illustrates the execution plan representation received by node 6. It is a serialised form of the execution plan shown in Fig. 3. If all constraints of this execution plan are satisfied, the *Execution* module starts execution at line 0. It queries *Remote Service* (which forwards the query to node 1) whether the stove is turned on. If the reply is positive, the branch at line 1 is not taken and execution continues at line 2. If the reply is negative, it takes the branch and execution continues at line 6; thus, it asks node 2 if the stove is turned on. If the execution reaches an *END* command, it succeeds and adds the state *stoveOn* to the network states. Execution is stopped if no command is left or if the evaluation of a condition is not successful and there is no branch statement.

3. RELATED WORK

Implementing sensor network applications by expressing global network behaviour is referred to as *macroprogram-*

ming in the wireless sensor network literature. An example is Kairos [3], which provides high-level abstractions for access to neighbour nodes in the network and for sharing data between neighbours. They show that these abstractions allow to write distributed applications in a centralised fashion. Compared to our approach Kairos is still *node-dependent.* In other words they provide explicit abstractions for nodes. Whereas our runtime system interprets tasks, their backend system generates the complete application binary; this makes it expensive to change the running application. In contrast to their imperative language the declarative nature of our approach allows to clearly separate the definition from the execution of an application.

In terms of the declarative nature of the language Regiment [11] is similar to our approach. Regiment is a functional language. They argue that functional languages are intrinsically more compatible with distributed implementation over volatile substrates than are imperative languages. Similar to Kairos Regiment is node-dependent; this gives them less flexibility in distributing tasks in the network. Furthermore RuleCaster's tasks can be much more specialised and different to each other, which is more suitable for a heterogeneous network infrastructure. In both systems — Regiment and Kairos — the programmer is required to define operations on the 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 of 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.

RuleCaster provides *node-independent* abstractions. A distributed sensor network application is expressed in a network independent way; there are no explicit abstractions for nodes. There has been some work in this direction based on SQL-like language abstractions [6, 16]. The sensor network is seen as a distributed database. SQL-like query statements can be used collect sensor data from the network. While these systems provide useful tools for collecting sensor readings, they do not focus on processing data in the network. The RuleCaster system collects data, interprets it and reacts to it in the sensor network.

The usefulness of providing semantic sensor data abstractions are pointed out in [5]. They present a service-based system. Low-level events (i.e. sensor data) are transformed into high-level semantic events by a composition of services. They propose a planner that automatically generates this service composition in order to answer a high-level user query [15]. This composition of services is similar to the hierarchical structure of rules used in RuleCaster. Their focus is on providing semantically useful sensor data to the user, whereas RuleCaster focuses on sensor data collection, processing and actuation in the network.

4. CHALLENGES

The novel nature of our approach brings a variety of new challenges with it. We address them in the following sections. We focus on the research questions. Some of them have been answered in earlier sections, others are open for future research.

4.1 Network Model

The network model is generated and dynamically updated by a network monitoring process. A lot of information is gathered from the nodes. This process has to be efficient; energy, for example, may be a scarce resource in some sensor networks. There can be other information sources apart from the nodes; well-known position of nodes and hardware specifications are two examples. All these sources have to be fused into one consistent network model.

In Sec. 2.2 we presented our network model representation. Is this representation useful? Wireless networks can be very dynamic. So the characteristics can quickly change; communication links, for example, appear and disappear. It is an open question whether and how uncertain information can be represented in the network model. This is related to the question about how often and when the network model has to be updated. Some aspects of the network are static (e.g. hardware), others are predictive (e.g. energy consumption), while some can be random. It is important to consider these aspects when deciding on an update strategy.

4.2 Compiler

There are a lot of possible distribution strategies. An optimal choice can depend on the network characteristics or the application requirements. It is important to base the distribution process on some well-defined criteria. Minimising energy consumption could be a possible criterion. It is an open question how these criteria can be clearly specified. We use the idea of a cost function; every operation (e.g. processing, communication) comes with a certain price. The distribution process tries to minimise the cost. The problem is that certain things are difficult to define in a cost function. Redundant processing, for example, might be desirable to increase overall stability of the system.

The network model is dynamic. At some point the compiler has to decide on a certain distribution of the tasks. In related work these dynamic network properties are described as *non-computational aspects of services* [5]. They suggest that the execution of services (i.e. tasks) must dynamically adapt to resource variations to preserve quality requirements. How dynamic adaptation can be described in the tasks and how it can be done is an open challenge and is highly related to challenges addressed in Sec. 4.3. A more in-depth discussion and proposal for adaptive resource allocation is given in [7].

It is clear that the distribution can affect the outcome of the application execution. By using redundant processing and communication, for example, reliability could be increased for a short period of time while the higher energy demands reduce the lifetime of the network. These aspects of an application must be conveyed to the application programmer so that the right design choices can be made.

Finally, the tasks have to be sent to the actual nodes in the network. Each node is assigned an individual task. Depending on the distribution strategy there might be a lot of redundant data if each task is individually sent to its respective node. A compression scheme based on similarities of tasks could make distribution more efficient with respect to energy consumption or time. The difference to previous work on code distribution (e.g. Deluge [4]) is that there is no application image that is the same for each node in the network. Others extended Deluge and focused on updating only those parts of the application image that have changed [8].

4.3 Runtime System

The runtime system poses a lot of difficult challenges. Group collaboration is a major one. The difficulty is to address a group of nodes and communicate with them reliably and efficiently. There is some prior art on group communication in wireless sensor networks [10, 14, 13]. These abstractions can build a useful basis for a runtime system. However, collaboration is more than just communication. It is about finding a solution under the given rule, spatial and temporal constraints.

The advantage of our approach is that lower-level details are hidden from the application programmer. The communication stack could be exchanged without directly affecting the application written in RCAL. Because of the spatial and temporal constraints in RCAL rules we expect characteristic communication patterns. This knowledge could be used in the design of specific (energy efficient) communication protocols used by the runtime system.

Wireless communication is very expensive in terms of energy consumption. In order to evaluate certain rules, a node has to use services offered by other nodes. A query/reply protocol seems to be the obvious choice for this kind of communication. However, the queries might be repeated several times and the reply might be the same. An event-based system is an alternative. However it is not clear if it is suitable for a dynamic and unreliable network. Some researchers propose the use of data models to predict remote sensor values in order to reduce communication [12, 2]. We are working on a semantic data level; the transformation from low-level data events into semantic data events can be seen as a data reduction technique. Thus, predictive techniques (or intelligent caching) could be an interesting approach to communication reduction because the space of possible data values is smaller.

An optimal distribution of tasks depends on the network model. The network can be very dynamic. Should it be the responsibility of the compiler to regenerate and update the tasks in the network? Or should the runtime system adapt the tasks to the changing environment? If the runtime system is allowed to adapt the tasks, we have to find a way to describe what it is allowed to do; each node should know what the overall application goal of the execution is.

An RCAL rule depends on several constraints (time, space and state). We have already talked about the communication aspect of the spatial constraint. The temporal constraint adds an additional dimension to the problem. Other researchers propose dynamic spatio-temporal data structures as an abstraction to this problem [11]. These abstractions are a very powerful theoretical construct for macroprogramming; however they fail to show how it could be implemented.

5. CONCLUSION

We implemented the RuleCaster compiler as a PC application. The runtime system was developed under TinyOS for the tmote sky platform [9]. In the current version we assume that the network model is given; we use a static description of the network as input to our compiler.

By having a distributed sensor network application automatically generated from a high-level implementation the application developer does not have to directly deal with communication, synchronisation or other low-level optimisation problems which make distributed applications complex and error-prone.

RCAL — a rule-based language — reduces the mental gap between the application domain model and the programming language. It makes it easier for an application developer who does not have the expert knowledge of the underlying network infrastructure to implement applications.

Sensor network applications are generally designed for a whole network. This notion should be reflected in the way a sensor network is programmed. We believe that providing abstractions and support that allow the programming of the network as a whole is a step into the right direction.

6. REFERENCES

[1] A. D. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. In *PERVASIVE*, 2006.

[2] S. Goel, T. Imielinski, and A. Passarella. Using buddies to live longer in a boring world. *percomw*, 0:342–346, 2006.

[3] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS*, 2005.

[4] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys'04*, 2004.

[5] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *BROADNETS*, 2005.

[6] 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.

[7] G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *NSDI*, May 2005.

[8] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: a flexible and efficient code update mechanism for sensor networks. In *EWSN*, 2006.

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

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

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

[12] D. Tulone and S. Madden. PAQ: Time series forecasting for approximate query answering in sensor networks. In *EWSN*, 2006.

[13] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

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

[15] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN*, 2006.

[16] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.