Programming the Ubiquitous Network: A Top-Down Approach

Urs Bischoff, Gerd Kortuem

Lancaster University, Lancaster LA1 4YW, UK

Abstract. A ubiquitous network is characterised by a large number of small computers embedded in our environment. Writing distributed applications for this kind of network can be a challenging task. Programming and re-programming each single device is not feasible; it is too time consuming, too costly and too error prone. We argue that a top-down approach to writing applications is useful. We propose a high-level language that can specify a ubiquitous network's global behaviour. A compiler can automatically generate device-level executables from this global specification.

1 Introduction

Writing and deploying software applications can be a challenging task. Especially when dealing with the ubiquitous computing vision of an "invisible" infrastructure embedded in our environment we face challenges we have not encountered in traditional networks. We are working on the idea of embedding computers and sensors into real physical objects. By giving them processing and sensing capabilities we can make these objects "smart". Our vision is a world of these objects and other more powerful devices (e.g. PDAs) to form a smart environment. We refer to such an environment as a *ubiquitous network*.

From a system's perspective we face one main problem, which is inherent to this new kind of network: how can we effectively implement an application for such a large number of ubiquitous devices? This new network allows us to realise new kinds of applications. What we need are suitable methods for designing, implementing and deploying applications.

With this large number of devices it is not possible to program and re-program each device individually. This would be too time consuming, too costly and too error prone. This is even more relevant if we consider a dynamic environment with changing device configurations and often changing requirements.

We argue that instead of focusing on each individual device independently we need to focus on programming the entire network as a whole. Knowing the individual device that executes the application is of less interest. We need to find ways to define applications independently of the underlying distributed computing environment and a way to inject them dynamically into the network.

We propose a solution that addresses the network as a whole. Rather than performing sensing and interpretation bottom-up, i.e. driven by available sensor data, we envision a top-down approach. This top-down approach is based

on three concepts: (1) a declarative model of the application, (2) a dynamic model of the ubiquitous network describing its capabilities and the location of its components and (3) a top-down approach for application logic checking that dynamically maps application tasks to the devices in the network.

2 The Top-Down Approach

Figures 1 and 2 illustrate the three core concepts and their relation to each other. The network model is the interface to the network. The application definition is built on top of this network model. A mapping process splits the application into a set of tasks. Each device's tasks are first translated into an intermediate representation: the execution plan. The execution plan is then translated into the running task. In this section we will describe these three concepts in more detail.

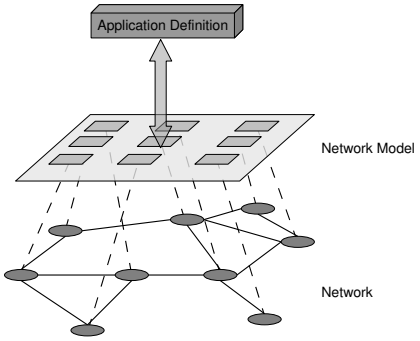

Fig. 1. The network model is the interface to the network. The application definition is built on top of this network model. A mapping process translates the application definition into a distributed application for the network given by the network model.

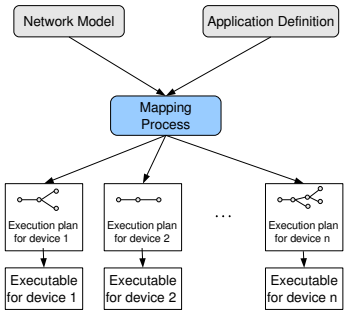

Fig. 2. A mapping process splits the application into a set of tasks. Each device's tasks are first translated into an intermediate representation: the execution plan. The execution plan is then translated into the running task.

2.1 Application Definition

We have developed a high-level specification language for 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 user expects. The declarative nature of this application language is essential because it allows us to seperate application logic from application

execution. The language is expressed in terms of rules. Rules are very natural - they can be understood by both humans and computers [3].

Example 1 shows a simple application definition. It defines an application that detects when a storage room is too hot. The network is in a state *tooWarm*

Example 1 An example of a simple application definition.

```
SPACE(storage), TIME(SIMULTANEOUS), STATE(normal) {
    STATE tooWarm :- product(X), hasToBeChilled(X), hot(), humid().
    hot() :- temperature(Y), Y>25.
    humid() :- humidity(Z), Z>90.
    ...
}
```

if there is a product X that needs to be chilled and the storage room is hot and humid. This rule has a spatial and a temporal constraint. The rule is only valid in a certain region and during a certain time interval. In our example, the rule is restricted to a region called *storage*. A rule consists of a goal (*tooWarm*) and several conditions (e.g. *product(X)*). The temporal constraint in our example specifies that all conditions have to be valid simultaneously in order to satisfy the whole rule. An application developer might have to specify several rules. In the example above there is a condition called *hot()*. However, it does not say how warm a room has to be in order to be hot. A rule specifying that it is hot if the temperature is more than 25 degrees centigrade has to be defined. Similar rules are defined for *product(X)* and *hasToBeChilled(X)*.

An essential part of our language are its space, time and state constraints. By using the temporal and spatial constraints we can define the global behaviour of a network. The state constraint can be seen as a pre-condition that has to be satisfied. In the given example the network has to be in the state *normal* for the rule with the goal *tooWarm* to be evaluated.

2.2 Network Model

The second part of our approach is the network model (cf. Fig. 1). 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 device 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 device is an example of an alternative source. Changing parts (e.g. energy level of devices) of this model are dynamically updated while static parts (e.g. hardware specification) are kept the same during the lifetime of the network.

Example 2 shows the interfaces (i.e. model descriptions) of the devices with *IDs 374* and *248*. Keywords are written in capital letters. Device *374*, for exam-

Example 2 Interface of devices 374 and 248.

```
STATIC PROPERTY(ID, 374);
STATIC PROPERTY(TYPE, "tmoteSky");
STATIC PROPERTY(CONNECTIVITY, 371);
DYNAMIC PROPERTY(CONNECTIVITY, 248);
DYNAMIC PROPERTY(ENERGY, 5);
DYNAMIC SERVICE("temperature", OUT);
DYNAMIC SERVICE("beep", IN);
STATIC SPACE("RoomD21");
```

```
STATIC PROPERTY(ID, 248);
STATIC PROPERTY(TYPE, "tmoteSky");
DYNAMIC PROPERTY(CONNECTIVITY, 125);
DYNAMIC PROPERTY(CONNECTIVITY, 374);
DYNAMIC PROPERTY(ENERGY, 3);
DYNAMIC SERVICE("humidity", OUT);
STATIC SPACE("RoomD21");
```

ple, can measure the temperature and therefore provides the service *temperature*. It is a *DYNAMIC* service that can return (*OUT*) one value. The keyword *DYNAMIC* means that it can return a different value at different times; in this example it tells you that the measured temperature can change. Actuators are also announced as services; *beep* is an actuator service that accepts one parameter (*IN*). The interface also provides some properties of the device. It shows that the current energy level is 5 and that the device has a static connection to device 371. Finally, the interface tells that the device is in *RoomD21*.

2.3 Mapping Process

The third core part of our approach is a process that dynamically generates the distributed application, i.e. it maps the application onto the network. Analogously to a compiler for a single device application we define a process that translates the rule-based application definition into a distributed application for the network given by the network model. Each device is assigned a task in the execution of the whole application.

Figure 2 illustrates this translation process. The compiler generates an individual task for each device. 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. An execution plan is a directed graph. Each node in this graph represents a condition that has to be satisfied in order to satisfy the whole rule. The execution plans are individually created for each device.

The rules do not specifiy where the tasks are evaluated. Several distribution strategies are possible: a centralised or a distributed solution are two examples. In the centralised solution, for example, only one device is responsible for all the processing; it collaborates with its peer devices by querying their services.

A basic example of the mapping algorithm is depicted in Fig. 3; it shows the generation of the execution plan for device *374*. A rule generally consists of a goal and several conditions. Each condition can be final or can be the goal of another rule. This produces a tree structure for each state rule. The mapping algorithm traverses this tree; if the given device provides a service for evaluating the condition represented by a node in this tree, it can execute this part of the rule (i.e. the whole subtree) locally. If there is any leaf node that represents

a condition that cannot be evaluated by the device's provided services (e.g. *humidity* is not provided by device *374*), it has to collaborate with other devices (e.g. device *374* collaborates with device *248*) in order to evaluate this condition.

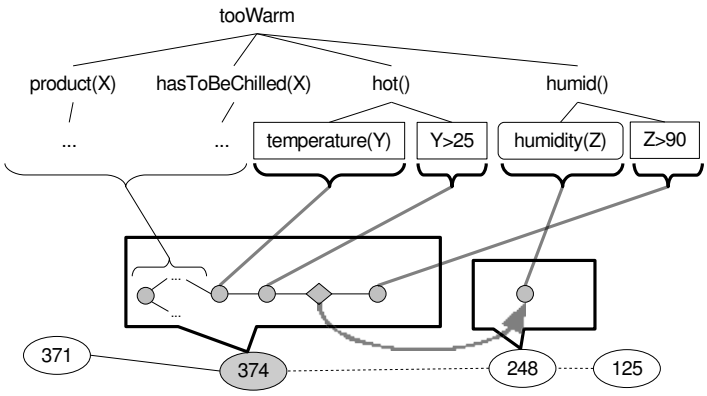

Fig. 3. The application definition is translated into a distributed application.

3 Related Work

Rule-based programming of ubiquitous networks has been introduced in related projects. Strohbach et al. propose a Prolog-like language that is used to write applications for a mote-like platform embedded in chemical containers [9]. Health and safety regulations can easily be translated into their rule-based language. In contrast to our network-centric view they focus on node-centric application development. Dey et al. found that the end-user likes to use rules to describe context-aware applications for a smart home environment [3]. Rules can be seen as an application model that is close to our mental model of the application.

Implementing sensor network applications by expressing global network behaviour is referred to as *macroprogramming* in the wireless sensor network literature. An example is Kairos [4], 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. In contrast to their imperative language the declarative nature of our approach allows to clearly seperate the definition from the execution of an application.

In terms of the declarative nature of the language Regiment [8] 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 still node-dependent; this gives them less flexibility in distributing tasks in the network. Furthermore our 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 our mapping process more freedom in assigning tasks to nodes.

Our approach provides *node-independent* abstractions. A distributed 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, 11]. 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. In-network processing is limited to filter and aggregation operations on raw sensor data; it does not deal with higher-level semantic data. Our system collects data, interpretes it and reacts to it in the network; this distributed in-network processing decreases communication cost and improves reliability [2].

The usefulness of providing semantic sensor data abstractions are pointed out in [5]. They propose a service-based system. A service is a component that simply transforms input events into output events, adding new semantic information as necessary. 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 [10]. This composition of services is similar to the hierarchical structure of our rules. Their focus is on providing semantically useful sensor data to the user, whereas our approach focuses on sensor data collection, processing and actuation in the network.

4 Current Status and Future Directions

The mapping process was implemented as a PC application. Addresses of collaboration peers are statically assigned at compile time. We are investigating more flexible addressing schemes that are more suitable for the dynamic nature of sensor networks. The current addressing scheme only makes absolute spatial constraints possible. We are investigating the use of relative spatial constraints (e.g. ask all nodes within a distance of 5m or all nodes that are in the same room).

The execution plan is an intermediate representation of a device's task. We have implemented a runtime system that interpretes execution plans on a device. The runtime system was developed under TinyOS [1] for the tmote sky

platform [7]. The current version only supports single hop communication. In parallel with the development of a different addressing scheme, we are working on a better abstraction layer for group communication. An important research direction is better support for temporal constraints. Distributed sensor events have to be fused in order to evaluate certain tasks; this requires synchronisation and timestamping of events. We want our system to be more flexible and support temporal constraints based on intervals. Our rules can specifiy that distributed events have to occur within intervals of 30s, for example, in order to satisfy the dependent rule. The evaluation of this kind of rules requires distributed reasoning techniques based on intervals; it is still an open question how this could be efficiently implemented.

In Sec.2.2 we introduced the network model and mentioned that it is automatically generated. We did not go into details how it can be done. In our future research we will investigate efficient and consistent ways to explore the services and properties of the devices in the network. Furthermore, we will develop methods to automatically configure devices with additional services based on their capabilities.

5 Conclusion

The proposed top-down approach is suitable for heterogeneous networks. This approach does not require all nodes to have the same common communication protocols. We still require them to be able to communicate with each other. However, we do not build applications on top of certain common layers.

By automating the translation process the application developer does not have to deal with communication, synchronisation or other low-level optimisation problems which make distributed applications complex and error-prone. This translation process is centralised, which is not a weak point of our approach. Applications are generally developed by a centralised entity; thus it makes sense that the mapping process (i.e. our compiler) generates the running application before it is distributed to the network.

Ubiquitous network applications are generally designed for a whole network. This notion should be reflected in the way a 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.

References

1. TinyOS community forum. http://www.tinyos.net, 2006.
2. S. Ahn and D. Kim. Proactive context-aware sensor networks. In *Wireless Sensor Networks (EWSN 2006)*, 2006.
3. A. D. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. In *PERVASIVE*, 2006.
4. R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS*, 2005.

5. J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *2nd International Conference on Broadband Networks*, 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. moteiv. tmote sky. http://www.moteiv.com, September 2005.
8. R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN*, 2004.
9. M. Strohbach, H.-W. Gellersen, G. Kortuem, and C. Kray. Cooperative artefacts: Assessing real world situations with embedded technology. In *UBICOMP*, 2004.
10. K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Wireless Sensor Networks (EWSN 2006)*, 2006.
11. Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.