A Code Generator for Distributing Sensor Data Models

Urs Hunkeler and Paolo Scotton

IBM Zurich Research Laboratory,
Säumerstrasse 4, 8803 Rüschlikon, Switzerland
{hun,psc}@zurich.ibm.com

Abstract. As wireless sensor networks mature, it becomes clear that the raw data collected by this technology can only be used in a meaningful way if it can be analyzed automatically. Describing the behavior of the data with a model, and then looking at the parameters of the model, or detecting differences between the model and the real data, is how experimental data is typically used in other fields. The work presented here aims at facilitating the use of sensor data models to describe the expected behavior of the sensor observations. The processing of such models can be pushed into the wireless sensor network to eliminate redundant information as early in the data collection chain as possible, thus minimizing both bandwidth requirements and energy consumption.

1 Introduction

Wireless sensor networks (WSNs) promise cheap sensor deployment to monitor an area of interest in great detail. WSNs are, for instance, being used to measure seismic activity at volcanoes [17] or the micro-climate of glaciers [3]. Such sensor network deployments generate data at a much greater spatial resolution than more traditional observation techniques such as wired networks or data loggers. In addition, the data generated by these networks is available for immediate use. However, the huge amount of data has to be processed. Nobody will look at every single sensor reading. Instead, the data is used to evaluate physical models of the observed phenomena, and to detect situations where such models do not represent the observed behavior accurately.

To illustrate this, let us consider a hypothetical sensor network deployment (partially based on a real case [2]). In this hypothetical deployment a mountain village experiences sporadic floods caused by a glacier. To predict floods and alert the population, climatologists install a sensor network to monitor the micro-climate of the glacier by observing the surface temperature of the ice, the duration and intensity of sunshine, the amount of precipitation, and other similar factors. Based on a model of the glacier's behavior the data from the WSN is used to predict floods. The model could describe how water accumulates, and under what conditions the ice barrier breaks and releases the water. The model will not accurately predict the behavior of the glacier if an unexpected event occurs. For instance, a nearby dirt avalanche could cause the glacier to be covered

S. Hailes, S. Sicari, and G. Roussos (Eds.): S-Cube 2009, LNICST 24, pp. 127–143, 2009.

with a small layer of dust. The dust could completely change the heat absorption rate of the glacier, and thus how much water is melted on a sunny day. This model rupture might be detected because the measured surface temperature of the glacier differs from the expected surface temperature.

To deal with the large amount of data generated by a WSN, it is necessary to use data models to simplify the analysis of this data. Currently, data models are processed on back-end systems. Many data models, especially if based on complex physical models, are computationally expensive and therefore cannot be processed efficiently on the low-power devices typically used for sensor networks. It is, however, possible to do a first part of the processing already within the WSN. In this way, only the data necessary for the model processing rather than every single sensor reading is transmitted. This helps both to reduce the power consumption and to resolve bandwidth bottlenecks. In addition, some data models are able to exploit redundancy in sensor readings to make the data assimilation of a sensor network more robust to transmission errors.

When using data models it is important to be clear about quality-of-information (QoI) needs. QoI has been defined [4] as a collection of attributes including timeliness, accuracy, reliability, throughput, and cost. A WSN has some obvious QoI characteristics, such as the amount of data to be transmitted and the measurement accuracy and frequency. Data models can increase the confidence in the data by combining information from multiple sensors. On the other hand, it is possible to increase the life-time of a WSN by allowing the QoI to be reduced.

We propose a framework that facilitates using data models to process sensor data, and that enables us to push part of the model processing into the WSN. The concepts behind this framework have been introduced in the positioning paper [11], which presented a model processing mechanism running entirely on the back-end system. In this paper we present a first implementation of a distributed model processing mechanism running partially inside the senor network, and in particular show how model descriptions can be compiled into a distributed program and how optimization techniques can be applied. The key contributions presented in this paper are: (1) the description of a framework to automatically process generic sensor data models, which also pushes part of the data processing into the WSN, (2) the presentation of the implementation of a concrete model called distributed linear regression, and (3) the lessons we learned by implementing this framework. Section 2 presents related work. Section 3 presents a data model based on linear regression that will be used throughout this paper to explain the concepts of the framework. Section 4 describes the network concept of WSNs and presents a concrete network topology that will be used in the examples. Section 5 introduces our model description language. Section 6 discusses distributed aggregation of linear functions. Section 7 explains how the distributed model-processing algorithm is generated. Section 8 describes the supporting services that are needed to run the model-processing algorithm. Section 9 presents the experience we gained by implementing the framework. Section 10 concludes the paper.

2 Related Work

TinyDB [14] is a framework based on TinyOS that lets users see the WSN as a database. Querying sensors results in data being acquired by the network. In some cases, queries using aggregation functions are calculated partially inside the network. TinyDB supports aggregation, energy-aware query constraints, and continuously running queries. However, TinyDB was never aimed at model-processing. The language is based on SQL and might not be intuitive for users of WSNs without a computer-science background. TinyDB is no longer maintained.

MauveDB [6] is an extension of Apache Derby, an open source relational database implemented in Java™[1]. MauveDB offers the user a novel kind of view that calculates its data based on a sensor data model. Currently, supported models are based on either linear regression or correlated Gaussian random variables. Model processing is done entirely on the back-end system.

In Distributed Regression [9], a model based on linear regression has been implemented to run entirely within the WSN. The observations are approximated with a base function that linearly combines model coefficients with functions of the query parameters. The network transmits the model coefficients describing the observations in the network to the sink. An application on the sink can then approximate values of the observations anywhere within the network. Using this linear regression model enables a significant reduction of the amount of data being transmitted in the network. The implementation is specific to and optimized for this type of models.

BBQ [5] implements a model based on multivariate Gaussian random variables that runs entirely on the back-end system. Sensor readings and correlations among the readings of different sensors are used to determine a query plan that uses the least amount of energy to gather just enough new information from the network to answer a query while respecting the error bounds given.

3 Linear Regression

The framework is designed to be applicable to a wide range of different sensor data models and processing algorithms. Throughout the remainder of this paper we will focus on linear regression [9] as our example to explain different concepts. In this section we introduce the model and show in detail how it can be applied to sensor data.

Linear regression is a method for finding a set of dependent variables such that the regression function best fits the data. Let $f()$ be the regression function, $x_1, \ldots, x_p$ the function arguments, $a_1 \ldots a_c$ the dependent variables, and $g_1() \ldots g_c()$ a set of functions that combine the arguments of the outer function. The linear regression function then has the basic form:

$$f(x_1, \ldots, x_p) = a_1 g_1(x_1, \ldots, x_n) + \cdots + a_c g_c(x_1, \ldots, x_n)\,. \tag{1}$$

[1] Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

Let a data set D be composed of tuples, and let a tuple $d_i \in D$ be composed of the actual value v_i and a set of meta-data $x_{i,1}, \ldots, x_{i,p}$:

$$d_i = \{v_i, x_{i,1}, \ldots x_{i,p}\} . \tag{2}$$

Linear regression finds the dependent variables $a_1 \ldots a_c$ such that the sum of the squared difference between the values in v_i and the corresponding values from the regression function $f(x_{i,1}, \ldots x_{i,p})$ is minimized. If D consist of k tuples $d_1 \ldots d_k$, linear regression finds

$$\underset{a_1,\ldots,a_c}{\operatorname{argmin}} \sum_{i=1}^{k} \left(v_i - f(x_{i,1}, \ldots x_{i,p})\right)^2 . \tag{3}$$

Let us use linear regression to model the temperature measured in a WSN as a function of the physical sensor locations and the measurement time. Let x and y be the Cartesian coordinates of the sensor's location (measured for instance in meters), and let t be the time of the measurement (for instance expressed in seconds since the start of the experiment). Throughout this paper, we will use the following function to model the temperature readings:

$$f(x, y, t) = a_1 + a_2 x + a_3 y + a_4 t + a_5 t^2 . \tag{4}$$

We call the linear regression function *model function*, as we use it to model the sensor data. Similarly, we call the dependent parameters $a_0 \ldots a_4$ *linear coefficients* or *model parameters*. The model function and the model parameters together fully define the model for a particular set of data. In our model, the functions $g_0(), \ldots, g_c()$ are

$$g_1(x, y, t) = 1 \tag{5a}$$
$$g_2(x, y, t) = x \tag{5b}$$
$$g_3(x, y, t) = y \tag{5c}$$
$$g_4(x, y, t) = t \tag{5d}$$
$$g_5(x, y, t) = t^2 . \tag{5e}$$

We define a query on the model to be equivalent to the evaluation of a model function with a set of arguments, and the set of arguments used in the query is called *query arguments*. In our example, the query arguments are x, y, and t. In most cases the model will not be perfect and will produce results that differ from the measured values. This modeling error is a measure of the ability of the model function to represent the data accurately.

Before the model can be used to answer queries, its parameters $a_1 \ldots a_5$ need to be determined. We call functions that determine the values of model parameters *learning functions*. To determine $a_1 \ldots a_5$ in our example, let S be a set of n sensors and for each sensor $s_i \in S$ let us consider a set of measurement values at times $t_1 \ldots t_r$ noted $\{v_{i,1} \ldots v_{i,r}\}$. In addition, for each sensor $s_i \in S$, let x_i

and y_i be its Cartesian coordinates. The model function and the measurements form the following equation system:

$$\begin{aligned} v_{1,1} &= u_1 + u_2 x_1 + u_3 y_1 + u_4 t_1 + u_5 t_1^2 \\ v_{1,2} &= u_1 + u_2 x_1 + u_3 y_1 + u_4 t_2 + u_5 t_2^2 \\ &\vdots \\ v_{2,1} &= u_1 + u_2 x_2 + u_3 y_2 + u_4 t_1 + u_5 t_1^2 \\ &\vdots \\ v_{n,r} &= u_1 + u_2 x_n + u_3 y_n + u_4 t_r + u_5 t_r^2 \,. \end{aligned} \tag{6}$$

This linear equation system can be written in matrix form:

$$\underbrace{\begin{bmatrix} 1 & x_1 & y_1 & t_1 & t_1^2 \\ 1 & x_1 & y_1 & t_2 & t_2^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_2 & y_2 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & y_n & t_r & t_r^2 \end{bmatrix}}_{H} \underbrace{\begin{bmatrix} u_1 \\ \vdots \\ \\ \\ \vdots \\ u_c \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} v_{1,1} \\ v_{1,2} \\ \vdots \\ v_{2,1} \\ \vdots \\ v_{n,r} \end{bmatrix}}_{\mathbf{v}} . \tag{7}$$

The factors of the linear equation system can be represented as a matrix H. The coefficients we would like to determine form the vector $\mathbf{u}$. The sensor readings are grouped into vector $\mathbf{v}$. The linear coefficients should be determined such as to minimize the overall error (see Equation 3). We can do this with the following equation:

$$(H^T H)\hat{\mathbf{u}} = H^T \mathbf{v} \,, \tag{8}$$

where $\hat{\mathbf{u}}$ represents the estimate of the linear coefficients minimizing the error. The matrix $\hat{H} = H^T H$ has the dimensions $c \times c$, where c is the number of unknowns in the equation system. This equation can easily be solved using Gaussian elimination.

The matrix $\hat{H} = H^T H$ and the vector $\hat{\mathbf{v}} = H^T \mathbf{v}$ have interesting properties that enable a distributed determination of their values. To simplify notations, let $\hat{g_k}(i,j) = g_k(x_i, y_i, t_j)$. Then the elements of $\hat{H}$ and $\hat{\mathbf{v}}$ are calculated with the following formulas:

$$\hat{H}_{l,m} = \sum_{i=1}^{n} \sum_{j=1}^{r} \hat{g_l}(i,j) \hat{g_m}(i,j) \qquad \forall l, m \in \{1, \ldots, c\}^2 \tag{9}$$

$$\hat{\mathbf{v}}_l = \sum_{i=1}^{n} \sum_{j=1}^{r} \hat{g_l}(i,j) v_{i,j} \qquad \forall l \in \{1, \ldots, c\} \,. \tag{10}$$

From Equation 9 it is clear that $\hat{H}$ is symmetric and thus only has $\sum_{i=1}^{c} i = \frac{c(c+1)}{2}$ unique elements. Consequently, the total number of unique elements from both $\hat{H}$ and $\hat{\mathbf{v}}$ that need to be known to solve the linear equation systems is

$$N_{tx} = \underbrace{\frac{c(c+1)}{2}}_{\text{for } \hat{H}} + \underbrace{c}_{\text{for } \hat{\mathbf{v}}} = \frac{c(c+3)}{2}\,. \tag{11}$$

Both $\hat{H}$ and $\hat{\mathbf{v}}$ are sums of elements only depending on information provided by a single sensor node. As we will see, sums are easy to aggregate, and breaking linear regression down in the way shown here is key to distributing the calculation of linear regression across nodes in the WSN.

4 Network Setup

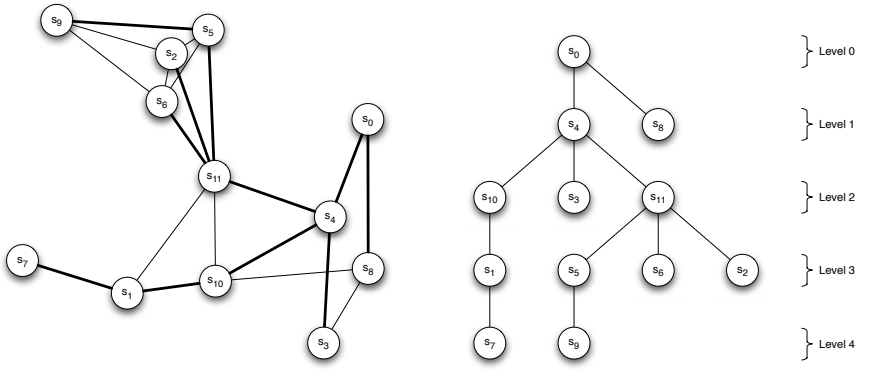
To illustrate the concepts in this paper we will use a sample network setup, which is defined and described next.

A WSN consists of *sensor nodes*, which have as basic components a number of sensors, a processing unit, a wireless transceiver, and some form of power supply. Sensor nodes are sometimes called *motes*. In our work, we use the Tmote Sky nodes (also known as TelosB) equipped with sensors measuring temperature, humidity and light. As software environment we use TinyOS [10]. Programs are written in NesC, which is a C-like programming language with WSN-specific extensions to easily modularize programming. TinyOS itself is an operating system for embedded sensor devices. As it is available in source code, users can modify every aspect of the system.

A WSN is a set of sensor nodes organized as a meshed network. If a radio connection can be established between two sensor nodes of the network, these two nodes are said to be *directly connected*. In the remainder of this paper, we will assume that connections between nodes are always bidirectional. This assumption is based on link quality measurements in [16] and on our own observations. For a given sensor node, we will call *neighbors* the set of sensor nodes to which this node is directly connected. We will assume that the sensor network is not partitioned and that it provides a routing mechanism: if two arbitrary sensor nodes are not directly connected, then they can communicate through other nodes in the network.

A complete system consists of the WSN and a back-end system connected through gateways. The back-end system might consist of several computers and software components that can communicate over a network different from the WSN, for example over a company's local area network (LAN). As a gateway is present in both the WSN and the back-end system, it needs to be able to communicate with both networks. This is typically solved by connecting a *gateway sensor node* to a *gateway computer* over a serial or USB cable. In this paper we will assume that only one gateway exists and that all components of the back-end system run on the gateway computer.

Typically, most communications in a WSN send observation data from every sensor to a data *sink*. The collection tree protocol (CTP) [7] in TinyOS is an example of a routing protocol that lets any node in the WSN send data to a sink. For the purpose of this paper, we only consider the case of a single sink in the



(a) A WSN topology. Thick lines are routes to the sink

(b) A hierarchical view of the WSN topology in Figure 1(a)

Fig. 1. A sample WSN configuration showing 1(a) the geographical distribution of motes and their connections, and 1(b) a hierarchical view of the routing tree

network. We further assume that the sink is also the gateway node connected to the back-end system. Routing algorithms based on the CTP principle operate as follows. The routing protocol establishes a spanning tree rooted at the sink and connecting all nodes in the network. When a sensor node wants to send data to the sink, it passes the data to its parent in the spanning tree. This operation is repeated by all parents of the node until the information reaches the sink. Therefore, to be able to communicate with the sink, a node only needs to know its parent in the spanning tree.

Figure 1(a) shows the geographic distribution of the sensor nodes in our sample WSN setup. The lines between the nodes indicate communication links; thick lines are links used for sending data towards the sink (s_0). Figure 1(b) shows the same network as a hierarchy of nodes. The top node is the sink. Nodes without children (s_2, s_3, s_6, s_7, s_8, and s_9) are leaf nodes. All other nodes relay messages from their children in addition to their own data.

5 Model Description Language

To describe sensor data models in an abstract way, we designed a model description language, whereby the aim was to design a language that is intuitive to use by people with little programming experience. For this reason we decided to develop a language similar to the mathematical languages used, for instance, in Matlab, SciLab or Octave. We presented this language in [11]. In this section, we explain the motivation behind these language design points that are specific to distributed model processing.

A model description essentially consists of the model function and the parameter-learning functions. In addition, it may contain configuration options

that determine, for instance, the applicable QoI parameters. The basic concept of the language is the sensor node. As models usually operate on a set of sensors, the model description language has to be able to express sets of sensors. As a starting point for doing this, let `S` denote the set of all sensor nodes in the network. In addition, each sensor node object also has a `neighbors` set and a number of associated sensors. The language uses the `forall` qualifier to apply a given expression to all elements in a given set. The `forall` qualifier allows the specification of additional constraints with an optional `where` clause.

Sensor readings can be accessed through sensor objects associated to a given node. For instance, if a sensor node has a temperature sensor, the current temperature value can be read with an expression of the form `sn.temp`. It is possible to access the n-th value in the past with the syntax `sn.temp[n]`. An index of 0 is equivalent to reading the current value. If values of two different sensor nodes are accessed, then an appropriate synchronization mechanism ensures that the sensors are sampled at the same time.

Mathematical operations clearly are an essential part of any model description. In addition to the basic mathematical operators, the model description language supports a number of special operators often used in model descriptions. This set of operators will be expanded in the future as need arises. Currently we have predefined the functions `sum`, `avg` and `LMS`, which are used to calculate the sum and average over a set of values, and the best fit of a function to a set of data, respectively. The principle of the `LMS` operator was discussed in Section 3, and its implementation is the subject of Section 6.

Model parameters and model functions are declared with an assignment using the equals sign (`=`). Model parameters describe the state of the model based on the measured sensor values. They can be global (the same value is shared in the entire network) or local (the value is only valid for a particular sensor node). In addition, a model parameter can be defined for a pair of sensors, for instance, to express the covariance of their readings. Model functions, in contrast, have a list of function arguments. The arguments qualify what exactly should be modeled, e.g., which sensor value should be modeled. As querying the model involves evaluating a model function, we call the function arguments *query parameters*. The model function definition on the right-hand side of the assignment involves a computation based on the model parameters.

In our sensor data model language, the model (Equation 4) can be expressed as shown in Listing 1. The learning function for the model parameters (based on Equation 3) is shown in Listing 2.

The computation of the model function is obvious, but the learning function does not appear in an explicit form. `LMS` stands for least mean squares. This operator calculates the coefficients for a linear regression function over a data set. `LMS` operates on the sensed values and the factors of the regression coefficients as expressed in Equation 4. As for this minimization problem we consider all sensors simultaneously, `LMS` takes as arguments vectors whose elements correspond to individual sensors. The first element in the vector is the actual value to be approximated by the linear regression. In this example, the value of the

Listing 1. Model Function

```
1 b(float x, float y, int t) = a[0] + si.x * a[1] +
2         si.y * a[2] + t * a[3] + t^2 * a[4];
```

Listing 2. Learning Function

```
1 a = LMS(forall si in S, t = 1 .. 5: si.temp[t],
2         1, si.x, si.y, t, t^2);
```

first element in the vector, `si.temp[t]`, corresponds to $s(x, y, t)$. The remaining elements are the factors with which the coefficients are to be multiplied. In this example, the first factor is the numerical constant 1, which means that the coefficient a_0 stands by itself. The second and the third factor, `si.x` and `si.y`, are the x and y coordinates of sensor s_i. The forth factor is simply the time t, and the fifth factor is the squared time, t^2. In our example each vector contains six elements, or five factors, which means that the linear regression function has five coefficients. Thus, the `LMS` operator will return a five-element vector.

A data set given as argument to the `LMS` operator usually consists of more than one vector. In the example above, the data set contains a vector for every sensor $s_i \in S$ and for every time $t \in \{1, 2, 3, 4, 5\}$. The actual temperature readings and the x and y coordinates are associated with s_i.

The sensor data model language is designed such that it can represent any mathematical closed-form expression. As the compiler needs to be able to determine the cost of calculating a sub-expression on a mote, the language is designed to be deterministic and does not support jumps and non-deterministic loops. More complex functionality can be achieved, if needed, by including additional elementary functions. These additional functions should be implemented such that the cost of computing them, or at least an upper bound for the cost, is known.

6 Aggregation and Linear Regression

Often an aggregate value over a set of sensor readings is desired, such as average, minimum and maximum values, and standard deviation. Madden et al. [13] describe aggregation in three steps: determining a *partial state record* for individual sensor readings by applying an *initializer* i, then combining these partial state records using a *merging function* f, and finally calculating the value of the aggregation using an *evaluator* e. Aggregations in which the size of the partial state record is significantly smaller than the original data set potentially enable a reduction of the amount of data to be transmitted in the network. Instead of transmitting and relaying every sensor reading in the network, nodes only transmit partial state records based on the data from their own sensor readings and the partial state records of their children. This is particularly interesting

for aggregations in which the size of the partial state record is constant, such as minimum and maximum values, averages, and sums.

The exact energy savings possible by using aggregation will have to be evaluated experimentally, as they strongly depend on the implementation details. In TinyOS, the energy consumption for message transmissions depends mainly on the number of messages sent rather than on the payload length of the messages. This is due to the default radio stack implementation, which senses the channel while waiting for a random back-off time prior to sending a message. As basis for comparing energy consumption, we take a very simple application that transmits a node's sensor readings using CTP [7]. CTP uses intermediate nodes to relay messages and does not alter these messages. For instance, in our network shown in Figure 1, the readings from sensor node s_7 would be relayed by the nodes s_1, s_{10}, and s_4 before they reach the sink s_0. Sending a message with readings from node s_7 results in a total of four message transmissions. Thus, if all nodes transmit their sensor readings, 28 messages are sent in the network.

With aggregation, each node only sends a single message that combines its readings with the readings of its child nodes. In our implementation we succeeded transmitting the partial state record of the `LMS` operator in a single message. Thus, each node waits for the partial state records of all its child nodes, combines the data, and then transmits a new combined partial state record to its parent. Sensor node s_{11}, for instance, sends a single message to its parent node s_4 instead of relaying the individual messages from nodes s_9, s_5, s_6, and s_2. Aggregating all the data within the network rather than transmitting every sensor reading results in only 11 message transmissions in our network. Aggregation thus can enable significant energy savings, especially in larger networks.

7 Compiling Models

To process a model, a program is generated that takes the sensor readings and calculates the model parameters. With this, queries from the user can be answered. In our example, a query is a call to the model function with specific values for the query parameters. The model function depends on the model parameters, which in this case are combined in a single vector `a` determined with the learning function. Once `a` is known, any query on the model can be answered. This section presents a method for generating the code to determine the model parameters in a partially distributed fashion.

A compiler takes a program as input, analyzes and transforms it, and produces the program in a different form as output. For our framework, the compiler reads a model description and produces code in the NesC and Java programming languages as output. To do this, the compiler reads the model description and forms an *internal representation* (IR) of the model by splitting the description into small pieces that each form a meaningful unit. Such units or *tokens* are, for instance, numerical constants, variable names, or mathematical operators. The compiler then analyzes and records the relationship between tokens. For instance, an operator operates on one or more input values and thus the token

associated with it has a relationship with the tokens that describe the operator's input values. Certain tokens, such as parentheses, only serve to determine the relationships between other tokens and can be discarded once all relationships have been established. The resulting representation of the model can be seen as a tree (shown in Figure 2(a)) in which the root node represents the model as a whole. The child nodes of the model node represent the different learning and model functions that together form the model. The nodes representing these functions in turn have child nodes that represent, in the case of model functions, the arguments to the functions, and that describe the mathematical expressions used to calculate the function. The representation of a program, or in our case a model, in such a tree form is called *abstract syntax tree (AST)*. We use Java Compiler Compiler (JavaCC [12]) to help us generate the code for reading and analyzing the model description. Additional information about compiling in general can be found in [1].

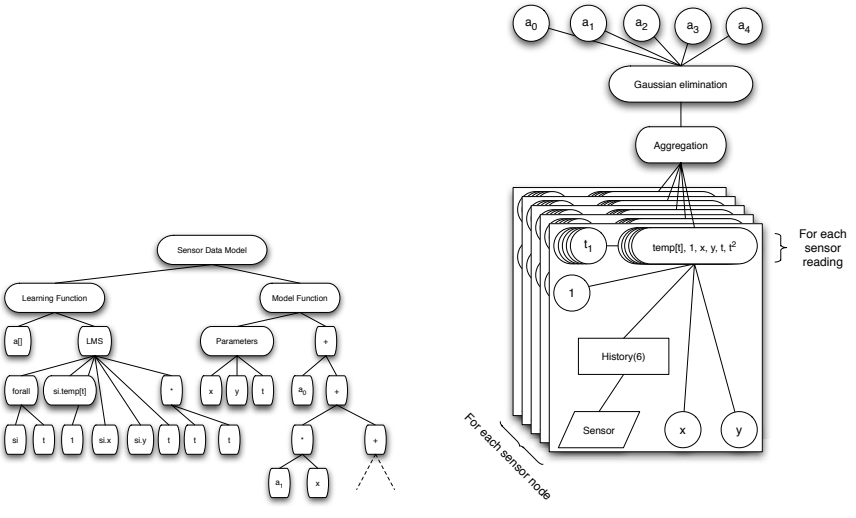


Fig. 2. (a) The linear regression model with (b) the learning function shown as abstract syntax trees. Note that because of space constraints not the entire model function is shown. In (b) the learning function has been arranged such that the multiplicity of the data sources and paths can be seen.

The basic AST plainly represents the model description. Before being able to generate distributed code to process this model, the AST must be augmented to include information related to specific nodes in the tree, such as the data type of each node. We distinguish between integer and floating point numbers, sensor nodes, sensors, vectors, and matrices. Vectors and matrices have associated dimensions and their elements in turn have associated data types. In the case of sensor nodes, the compiler needs to know which sensors are actually used

and what information needs to be stored on the nodes. In addition, we need to determine how many sensor readings have to be retained. To do so, we start by determining the data types of the leaf nodes, and then work our way back up the tree. For constants and sensors, the data type is clear. Before we can determine the data type of a variable or model parameter, however, we need to determine the data type of the expression defining this variable or parameter. The data type of an expression is typically based on the data types of its arguments. Also, if all arguments of an operator are constants, the result of the computation represented by this operator can be calculated at compile time, and the operator can be replaced by a constant. Similarly, if a variable is assigned a constant, all of its occurrences can be replaced by this constant. Sometimes, information in one part of the tree affects a completely different part of the tree, for instance, when the data type of a variable is determined in one place but the variable is also used in a different place. It is thus possible that not everything can be determined in one pass. Therefore, we reiterate the passes for as long as there still exists information to be determined and new information is still obtained in every pass. If no new information is obtained, but not everything has been determined, the compilation process is aborted with an error.

A fundamental aspect of model processing is to get the source data (e.g., sensor readings). The compiler determines which sensors are accessed on a node and then includes the appropriate code to sample the sensors. For instance, the expression `si.temp[t]` in the learning function in Listing 2 tells the compiler that the temperature sensor is accessed. If an element of a sensor node is accessed, and that element is not a known sensor, the compiler assumes it to be a variable associated with the sensor node. For instance, the expressions `si.x` and `si.y` in the learning function do not refer to any known sensors. As no method has been defined to determine the values for `x` and `y`, the compiler adds the necessary code for the execution environment to configure these values (see Section 8).

For every sensor used by the model, the model processing code will reserve memory for storing a history of the sensor readings. By default, the history size is 1, which means that only the last (current) sensor reading is stored. The current sensor reading is accessed from the model either by simply accessing the sensor object (e.g., `si.temp`) or by specifying 0 as the time value (e.g., `si.temp[0]`). Older readings are accessed by specifying a time value greater than 0. To determine the amount of memory to be reserved for storing the sensor readings' history, the compiler analyzes how the sensor readings are accessed. In our learning function (Listing 2), the sensor readings are accessed with the expression `si.temp[t]`. The compiler then analyzes the potential values that `t` can take. The variable `t` is defined in the context of a `forall` statement, which declares `t = 1 .. 5`. Thus in the example above the compiler deducts that the values for `t` vary between 1 and 5, and therefore reserves memory space for 6 sensor readings[2].

With the information we extracted from the model definition so far, it is straightforward to generate code that takes sensor readings as input and com-

[2] The compiler also needs to store the current sensor reading.

putes the results for a given query. All that needs to be done is to calculate the expressions and update the model parameters. Our framework can produce code that reads sensor data from a variety of different sources and answers queries. We compared two different statistical sensor data models and published our findings in [11]. The next step is to generate code that can run in a distributed environment, such as a WSN. To do this, the compiler needs to determine for each node in the AST whether the node is to be processed in the network or on the back-end system.

Sensors obviously have to be sampled on the sensor nodes. Currently, our framework simply stores all sensor readings retained and all node variables on the sensor nodes themselves. Constants are located in the same place as the operator accessing them. This reduces the problem to determining where to locate the operators such that the overall energy consumption in the WSN is minimized. Once data has reached the back-end system, it makes little sense to send it back into the WSN to process it further. Thus, operators that are closely associated with sensor nodes are more likely to minimize overall energy consumption if they are also located in the WSN. If there is an aggregation operator somewhere in the data flow, then in most cases the optimal approach is to process all operators between the sensor readings and the aggregation inside the WSN, to perform a distributed aggregation, and then to compute the remaining operators on the back-end system. Therefore, the framework currently focuses on finding an aggregation operator, and then using it to separate the AST into a part to be processed in the WSN and a part to be processed on the back-end system. The operator placement for our model is shown in Figure 2(b). Every sensor node executes the code in the rectangle in the lower part of the AST (this multiplicity is indicated by overlapping rectangles). For a specific number of past sensor readings, a port of this code execution is repeated (again, the multiplicity is indicated in the graph). The aggregation is distributed among the nodes, and Gaussian elimination takes place on the back-end system.

Once the elements of the AST that should run in the WSN have been determined, the corresponding code has to be generated. The framework bases itself on TinyOS and thus generates code in the NesC programming language [8]. Besides generating the node processing code for the WSN, it also includes the appropriate modules from the execution framework (see Section 8). In particular, it includes code to allow the execution framework to configure node variables (such as the x and y coordinates for the linear regression model presented in Section 5), and the code to access the communication modules. The communication methods supported currently allow data to be sent to the sink as-is or perform distributed aggregation of the data.

The framework generates Java code for that part of the model processing that is performed on the back-end system. It includes the appropriate interface code to communicate with the WSN. The code generated offers a dynamic interface to the application to specify the query and change the configuration of the nodes. However, at this stage the code is not self-sufficient and needs an appropriate execution environment.

8 Execution Framework

While testing our framework with the linear regression model, we found that distributed model processing needs basic support services. Every sensor network needs to transmit data to a sink. If the network performs aggregation, then nodes need to know their parents and potentially also their children. For many applications, the physical position of each node needs to be known. To demonstrate the proper operation of the linear regression model, we thus implemented a configuration mechanism, a tree routing algorithm for aggregation, and a simple time synchronization method.

We implemented a configuration service that enables the setting of parameters for a particular node. With this service, we configure the x and y coordinates of the physical location of a node prior to starting the model processing. An alternative would be to estimate the physical location during run-time (see, for example, [15]). We decided to use a configuration service rather than a locationing service, as for most locationing algorithms some nodes need to know their position in advance and therefore still would have to be configured.

When compiling a model, the framework assumes that any variable for which no explicit means to determine its value exists, is a configuration parameter. It will then generate a configuration message type with fields for all parameters. An application can set a configuration parameter for a particular node through the framework. The framework checks the parameter name against the list of configuration parameters. If the parameter name specified is valid, the framework will set the corresponding field, include the identity of the targeted node in the configuration message, and broadcast the message in the network.

Data collected in a sensor network needs to be routed to a sink. To do this, WSNs form a collection tree. When processing a model instead of simply collection raw data, the data is often aggregated within the network. The routing structure for a network that aggregates the data is essentially the same as for a WSN collecting raw data, as the data still needs to reach the sink. The difference is that every node, instead of relaying the data of its children as-is, aggregates its own data with that from its children before sending the data to its parent (see Section 6). This means that for one data-collection epoch each node sends exactly one message to its parent. We call this setup an *aggregation tree*.

We use the collection tree protocol (CTP) [7] in TinyOS to establish the routing tree. Instead of letting the collection tree forward messages automatically, we intercept each message and signal that the message should not be forwarded. We aggregate the information in the message with the node's own information and the information received from the other children. The information is then sent to the node's parent.

Before sending data to the parent, a node has to receive data from all its children. To do this, a node could keep track of its children and which ones already sent data in the current epoch. Once all children have sent their data, the node in turn sends its data to its parent. The version of CTP provided in TinyOS does not maintain a list of a node's children. Also, with this method it would be difficult for a node to predict when a child node is ready to send data,

which in turn makes it difficult for nodes to turn their radios off to save energy. Therefore, we adopted a time-synchronization strategy.

Synchronizing the clock of the nodes can be achieved by a variety of different protocols and algorithms. We found that one of the simplest approaches is also well suited to minimize energy consumption as determining required active periods for the radio is straight forward. In our implementation the network is synchronized by broadcasting the time from the sink node to the leaves of the tree. Based on the common view of the time, all nodes start the epoch at the same point in time. The nodes furthest down in the tree (level 4 in our sample network in Figure 1(b)) start by sending their data to their parents. After a fixed timeout, the nodes in the next higher level in the routing tree assume that all their children have sent their data, and send their aggregated data to their parent, until the data finally reaches the sink. This approach is simpler than explicitly waiting for data from all children, as nodes do not need to maintain a list of children. As the time period in which a node can expect transmissions from its children is well defined, nodes can turn off their radio when they do not expect transmissions, and thus achieve significant energy savings.

The basic services presented here are sufficient to implement the distributed linear regression model with the help of our framework. Other models might require additional services. A service can have multiple implementations, for instance, to optimize for speed, latency, reliability, or energy savings. We currently implemented very simple services as a proof of concept. Our framework facilitates uniting contributions from experts in different fields.

9 Results and Future Work

We implemented a framework for distributedly processing generic sensor data models. In this paper we focused on the model called distributed linear regression. The framework can be used for other models and we have, for instance, successfully implemented a model based on multivariate Gaussian random variables, which was inspired by [5].

The framework consists of the model description language, the compiler to generate the distributed code, and the execution framework that enables the code to run. During the implementation and testing process, we refined the model description language to make it easier to compile and also render it more intuitive to program. For instance, we removed a separate keyword for specifying model parameters. Instead, the compiler now recognizes model parameters by the form of their definition.

The implementation of the compiler also elucidated the key differences between normal programs and distributed WSN programs. Whereas for traditional ASTs it is sufficient to show the dependencies of nodes, for the distributed approach the dependencies themselves have properties that describe how the information flowing from the child to the parent is communicated and what the associated cost is. The main challenge in compiling distributed programs lies in optimizing this communication.

After having generated the distributed code, we realized that prior to running it we needed an execution framework for configuring the nodes, and for handling the communication between network and gateway. Especially communicating floating point values was challenging, as TinyOS does not directly support them. We solved this issue by copying a memory image of the variables holding floating point values. The `Float` class in Java has methods to convert between floating point numbers and byte arrays. Fortunately, the floating point representation of the TinyOS devices is compatible with Java's byte array representation.

Running linear regression as a distributed model in the current execution framework confirmed that the algorithm was running properly. However, sometimes CTP delayed messages, such that with the staged aggregation not all messages were received on time. We will have to analyze the exact reasons for this behavior. We will also implement our own version of a tree-routing algorithm that will enable us to turn off the radio when the mote is not expecting any messages. This will enable us to experimentally confirm the energy savings of our approach.

10 Conclusion

In this paper we presented an integrated approach for distributing the calculation of sensor data model processing. Our framework consists of a model description language, a compiler for this language that generates distributed code, and an execution framework needed for running the distributed programs. The framework is extensible and open for contributions from experts in very diverse fields. In contrast to previous work, which concentrated on studying a specific aspect of model processing and then optimizing it, our work is, to the best of our knowledge, the first holistic approach to generic model processing.

References

1. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press, New York (2003)
2. Barrenetxea, G., Ingelrest, F., Lu, Y.M., Vetterli, M.: Assessing the challenges of environmental signal processing through the SensorScope project. In: Proceedings of the 33rd IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2008), pp. 5149–5152 (2008)
3. Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M., Couach, O., Parlange, M.: SensorScope: Out-of-the-box environmental monitoring. In: Proceedings of the 7th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2008), pp. 332–343 (2008)
4. Bisdikian, C.: On sensor sampling and quality of information: A starting point. In: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW 2007), pp. 279–284 (2007)
5. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J.M., Hong, W.: Model-driven data acquisition in sensor networks. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004), pp. 588–599 (2004)

6. Deshpande, A., Madden, S.: MauveDB: Supporting model-based user views in database systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD 2006), pp. 73–84 (2006)
7. Fonseca, R., Gnawali, O., Jamieson, K., Kim, S., Levis, P., Woo, A.: The collection tree protocol (CTP), version 1.8 (February 2007), http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html
8. Gay, D., Welsh, M., Levis, P., Brewer, E., Von Behren, R., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI 2003), pp. 1–11 (2003)
9. Guestrin, C., Bodik, P., Thibaux, R., Paskin, M., Madden, S.: Distributed regression: An efficient framework for modeling sensor network data. In: Proceedings of the Third International Symposium on Information Processing in Sensor Networks (IPSN 2004), April 2004, pp. 1–10 (2004)
10. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. ACM SIGPLAN Notices 35(11), 93–104 (2000)
11. Hunkeler, U., Scotton, P.: A quality-of-information-aware framework for data models in wireless sensor networks. In: Proceedings of the First International Workshop on Quality of Information in Sensor Networks (QoISN 2008), September 2008, pp. 742–747 (2008)
12. JavaCC - a parser/scanner generator for Java (November 2008), https://javacc.dev.java.net/
13. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: A tiny aggregation service for ad-hoc sensor networks. SIGOPS Oper. Syst. Rev. 36, 131–146 (2002)
14. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: The design of an acquisitional query processor for sensor networks. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003), pp. 491–502 (2003)
15. Savarese, C., Rabaey, J.M., Beutel, J.: Locationing in distributed ad-hoc wireless sensor networks. In: Proceedings of the 2001 International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001), May 2001, pp. 2037–2040 (2001)
16. Srinivasan, K., Levis, P.: RSSI is under appreciated. In: Proceedings of the Third Workshop on Embedded Networked Sensors (EmNets 2006) (May 2006)
17. Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., Welsh, M.: Fidelity and yield in a volcano monitoring sensor network. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006), pp. 381–396 (2006)