UNIVERSITÀ DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica
Within European Masters in Informatics

Tesi di Laurea

High Level Programming of Wireless Sensor Networks Using Distributed Abstract Data Types

Relatori	Laureanda
Gian Pietro Picco	**Galiia Khasanova**
Klaus Wehrle	

Anno accademico 2007/2008

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Galiia Khasanova*)

Acknowledgements

Bla

Definitions and Acronyms

ADT	Abstract Data Type
CLDC	Connected Limited Device Configuration
DADT	Distributed Abstract Data Type
Java ME	Java Micro Edition
JiST	Java in Simulation Time
JVM	Java Virtual Machine
LN	Logical Neighborhoods
MAC	Media Access Control
MEMS	Micro-Electro-Mechanical Systems
OS	Operating System
QoS	Quality of Service
RF	Radio Frequency
Sun SPOT	Sun Small Programable Object Technology
SWANS	Scalable Wireless Ad hoc Network Simulator
VM	Virtual Machine
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actor Networks

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it [Mark Weiser]

1.1 Motivation

Recent advances in Micro-Electro-Mechanical Systems (MEMS) technologies and wireless communication have made it possible to deploy networks of sensor nodes called Wireless Sensor Networks (WSNs) that contain a large number of sensor nodes in several disparate environments. Each individual node is a multifunctional device characterised by its low cost, low power, and small form factor. They can communicate untethered across short distances using a variety of means, including Radio Frequency (RF) communication [3].

WSNs are currently used in a wide range of applications including health monitoring, environment monitoring, data acquisition in dangerous environments, and target tracking.

However, programming WSNs is currently a complex task that requires an understanding of the inner operation of WSNs. This limits its applicability in widely disparate fields by scientists unaware of the technology underlying WSNs. To deal with this problem, it is necessary to abstract the details of WSN operation thereby allowing the application programmer to develop WSN applications in spite of being an expert embedded system programmer. This is achieved using programming abstractions wherein the WSN application is viewed as a system and sensor nodes and sensor data are abstracted.

This work underlies the extension of a distributed programming abstraction called Distributed Abstract Data Types (DADTs) [24] that use high-level programming constructs to abstract interfaces to individual entities in a distributed network and allow the application programmer to communicate directly with the the entire distributed network.

Though DADTs appear ideal to the problem domain of abstracting WSN programming, no work had been performed on examining their suitability, and subsequently extending them for use in WSN applications. Additionally, a robust routing mechanism that allows for the selection of a subset of the sensor nodes in a WSN is required - a requirement that is not met by conventional WSN routing algorithms.

1.2 Contributions

This work makes the following contributions:

- It extends the DADT prototype developed in [24] to WSNs.
- It integrates the DADT mechanism in the application layer with a Logical Neighbourhood [19] routing mechanism that allows for the partitioning of the WSN network on the basis of neighbourhoods defined by logical predicates in the network layer.
- It evaluates the suitability of the developed prototype in simulated environment as well as in real-world wireless sensor networks.

1.3 Outline

The outline of the thesis is as follows: Chapter 2 presents an introduction to wireless sensor networks including descriptions of example WSN applications, provides details about selected hardware platforms and introduces the protocol stack used in a typical WSN application, coupled with a brief overview of underlying routing mechanisms for WSNs. Besides, it provides description of WSN programming models. Chapter 3 discusses the background underlying this work, and includes brief descriptions of DADTs, the LN routing mechanism, and provides a brief description of the structure of the DADT prototype that was modified during the course of this work. Chapter 4 presents the simluation environment and the hardware platform used to evaluate the prototype produced as part of this work. Chapter 5 describes the high-level design details of the implementation of the prototype. Chapter 6 provides implementation details and discusses the issues concerning testing of the prototype in simulated and in real-wolrd environment. Chapter 7 describes the metrics and experimental methodology used to evaluate this work. Finally, Chapter 8 provides the conclusions and possible avenues for future work.

Chapter 2

Introduction to Wireless Sensor Networks

This chapter presents a brief introduction to Wireless Sensor Networks (WSNs). Additionally, it describes the reference WSN protocol stack used as part of this work, and provides a short overview of WSN routing techniques.

2.1 Sensor Nodes

WSNs are, as described earlier (see Chapter 1), networks of sensor nodes, and are typically deployed randomly in a possibly large area where phenomena are required to be monitored.

As presented in Figure 2.1, a sensor node consists of the following elements:

- *Sensing unit*, which is comprised of a number of sensors and analog-to-digital converters.
- *Transceiver*, which facilitates node-node communication using a variety of techniques.
- *Processing unit*, that comprises a microcontroller/microprocessor that performs processing, and is associated with a storage unit.

- *Power unit*, which provides the energy required to run the sensor node, and can use chemical batteries or power scavenging units such as solar cells.

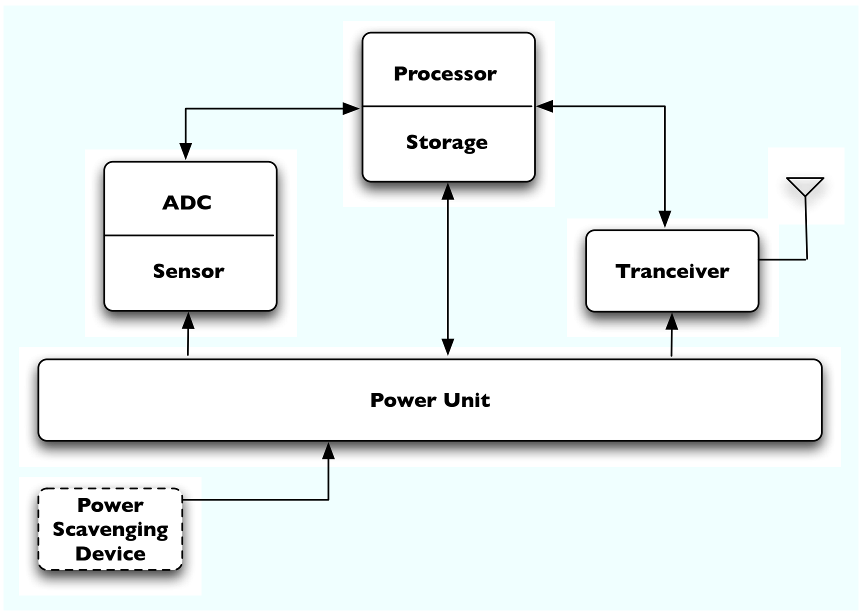

Figure 2.1: Architecture of a sensor node (adapted from [3]).

Due to the small size of the devices, sensor nodes have a number of constraints which affect the WSN built on top of it. These include [29]:

- *Power consumption constraint*, due to the fact that sensor nodes have limited energy supply. Therefore, energy conservation is the main concern when WSN applications are implemented.

- *Computation restriction*, caused by the limited memory capacity and processing power available on the sensor node. This places serious limitations on the use of data processing algorithms on a sensor node.

- *Communication constraint*, as a result of the minimal bandwidth available and a limited Quality of Service (QoS) provided by the sensor node's hardware.

Additionally, as the deployment of sensor nodes in the WSNs should be cost-effective, the cost of a single device is a supplementary constraint.

A WSN is self-organising system, given the random nature of the deployment. Its topology is subject to change because of characteristics of the wireless medium, and therefore, sensor nodes should be capable of dealing with changes of this kind in order to cope with hostile operating conditions, the failure-prone nature of sensor nodes and the possibility of redeployment of additional sensor nodes at any time during operation.

2.2 WSN Protocol Stack

The WSN protocol stack presented in [3] is an adaptation of a generic protocol stack [4].

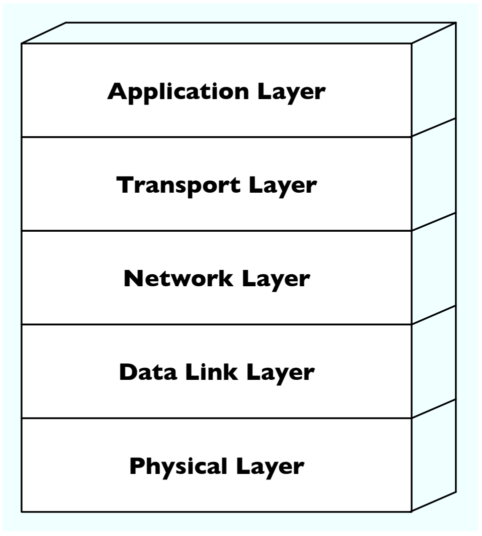

Figure 2.2: WSN protocol stack (reproduced from [3])

According to [3] the WSN protocol stack consists of the following layers:

- *Physical Layer*, which provides the transmission of data over the physical transmission medium.

- *Data Link Layer*, which deals with power-aware Medium Access Control (MAC) protocols that minimise collisions and transceiver on-time.
- *Network Layer*, which is primarily responsible for routing data across the network.
- *Transport Layer*, which provides reliable delivering of data and supports error checking mechanisms.
- *Application Layer*, where the application software resides.

The work presented here is resided fully on the application layer of the protocol stack, but uses network layer .. .

2.3 Programming Models for WSNs

Current WSN programming paradigms are predominantly node-centric, wherein applications are monolithic and tightly coupled with the protocols and algorithms used in the lower layers of the protocol stack. The main reason for this is the limited resources available on the sensor node, as was previously discussed in Section 2.1.

The primary problem with a node-centric approach is that most WSN applications are developed at an extremely low level of abstraction, which requires the programmer to be knowledgeable in the field of embedded systems programming. This stunts the growth in the use of WSNs in the large space of application domains where it may potentially be of use [22].

To increase the ubiquity of WSN usage, it is essential that protocols and mechanisms underlying WSN development recede to the background, and the application programmer is empowered to develop WSN applications at a higher level of abstraction. This can be achieved using programming models that engineer a shift in focus towards the system and its results, as opposed to sensor node functionality itself [22].

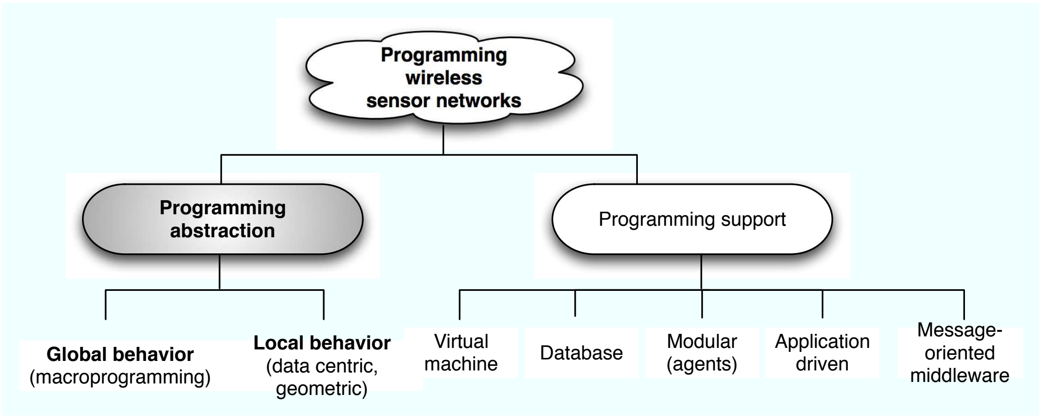
Programming wireless sensor networks
Programming abstraction
Programming support
Global behavior (macroprogramming)
Local behavior (data centric, geometric)
Virtual machine
Database
Modular (agents)
Application driven
Message-oriented middleware

Figure 2.3: Taxonomy of WSN programming models (reproduced from [14])

According to Yu et al [30], the use of such programming models is beneficial for WSN applications because:

- The semantics of a WSN application can be separated from the details of the network communication protocol, OS implementation and hardware.
- Efficient programming models may facilitate better utilisation of system resources.
- They facilitate the reuse of WSN application code.
- They provide support for the coordination of multiple WSN applications.

2.3.1 Taxonomy of WSN Programming Models

Existing programming models for WSNs cover different areas and can serve several different purposes. They can be classified into two main types, depending on the applications they are used for [14] (see Figure 2.3):

- *Programming support*, wherein services and mechanisms allowing for reliable code distribution, safe code execution, etc. are provided. Some examples of programming models that take this approach include Mate [16], Cougar [8], SOS [15], and Agilla [10].

- *Programming abstractions*, where models deal with the global view of the WSN application as a system, and represent it through the concepts and abstractions of sensor nodes and sensor data. Some examples of programming models that take this approach include Kairos [12], and EnviroTrack [2].

This section further discusses and classifies a subset of WSN programming models, namely WSN programming abstractions.

Programming abstractions may be distinguished depending on the way WSN nodes are being programmed, and therefore either be *global* (also referred to as macroprogramming) or *local* [14].

In the former case, the sensor network is programmed as a whole, and gets rid of the notion of individual nodes [22]. Examples of macroprogramming solutions include *TinyDB* [18] and *Kairos* [12].

In the latter case, the focus is on identifying relevant sections or *neighbourhoods* of the network. It is to be noted that these neighbourhoods need not necessarily be physical. The framework used and developed during the course of this work belongs to the latter class of programming abstractions.

Programming abstractions may also be classified on the basis of the nature of the language constructs made available to the WSN programmer [22].

Classification of WSN programming abstractions w.r.t. language constructs is presented in Figure 2.4.

The rest of this section discusses each of these bases for classification in detail, and is based on the work described in [22] unless explicitly mentioned otherwise.

2.3.1.1 Communication Span

The *Communication span* enabled by a WSN programming interface is defined as the set of nodes that communicate with one another in order to accomplish a task. The communication span provided by a given abstraction can be [22]:

- *Physical neighbourhood*

- *Multi-hop group*

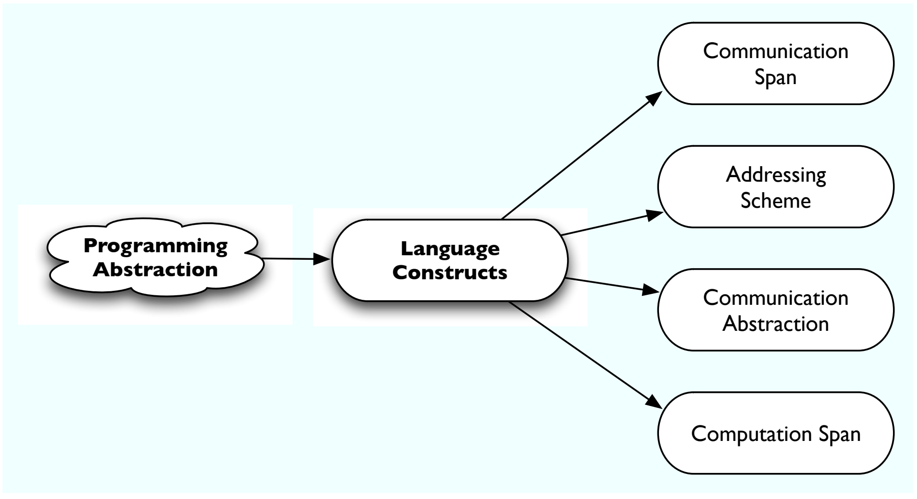

Figure 2.4: Classification of Programming Abstractions (adapted from [22])

- *System-wide*

Abstractions that use *physical neighbourhood* approach provide the programmer with constructs to allow nodes to exchange data with others within direct communication range. Examples of communication in physical neighbourhood include nesC [11] and Active Messages [9].

Abstractions with *multi-hop group* approach allow the programmer to exchange data among subsets of nodes in the WSN using multi-hop communication. These sets may either be *connected*, wherein there always exists a path between any two nodes in the set, or *non-connected/disconnected*, where "no assumptions on the geographical location of nodes belonging to the group" [22] can be made. EnviroSuite Framework [17] represents an example of connected multi-hop communication, and Logical Neighborhoods [20] - an example of non-connected multi-hop communication.

System-wide abstractions let the programmer use constructs that allow data exchange between any two nodes of the entire WSN. This may be seen as an extreme manifestation of the *multi-hop group* approach mentioned above. An example of system-wide communication is TinyDB [18] project.

2.3.1.2 Addressing Scheme

The *addressing scheme* specifies the mechanism by which nodes are identified. Typically, there are two kinds of addressing schemes used [22]:

- *Physical addressing*
- *Logical addressing*

In *physical addressing* schemes nodes are identified using unique identifiers. The same address always identifies the same node (or nodes, if duplicate identifiers exist) at any time during the execution of the application. The paradigmatic example of physical addressing scheme is Active Messages communication [9].

When a *logical addressing* mechanism is used, nodes are identified on the basis of application-level properties specified by the application programmer. Therefore, the same address, i.e. set of application-level predicates, can identify different sets of nodes at different times. An example of logical addressing is Logical Neighborhood communication [20].

2.3.1.3 Communication Abstraction

This classification basis defines the degree to which details of communication in a WSN are hidden from the application programmer's view. Programming interfaces may provide either [22]:

- *Explicit communication* primitives where the programmer working in the application layer has to handle communication aspects such as buffering and parsing.
- *Implicit communication*, where the programmer is unaware of the details of the communication process, and communicates using high-level constructs.

Active Messages [9] present an example of explicit communication approach. In contrast, the Active Regions programming framework [28] belongs to the latter category.

2.3.1.4 Computation Span

The *Computation span* enabled by a WSN programming interface is defined as the set of nodes that can be affected by the execution of a single instruction. The computation span provided by a given abstraction can be [22]:

- *Node*, when the effect of any instruction is restricted to a single node.
- *Group*, where the programmer is provided with constructs that could affect a subset of nodes.
- *Global* present an extreme case of previous type, a single instruction can impact every node in the WSN.

An example of node computation is the ATaG framework [5]; the Regiment system [23] implements group communication approach. The TinyDB [18] represents an example of global computation span.

2.3.2 Programming Models on the WSN Protocol Stack

WSN programming models often are placed between the application layer and the transport layer in the protocol stack shown in Section 2.2. As it can be seen from Figure 2.5, fine-grained details are hidden from the application programmer's view. These include:

- Higher-layer services such as routing, localisation, and data storage mechanisms (and optimisations).
- Lower-layers such as the MAC protocol used, and the physical means of communication such as RF communication.

2.4 Summary

This chapter presented an introduction to WSNs and discussed in detail the sensor nodes that constitute them. This was followed by a description of the WSN protocol stack. The chapter concluded with a classification of existing WSN routing

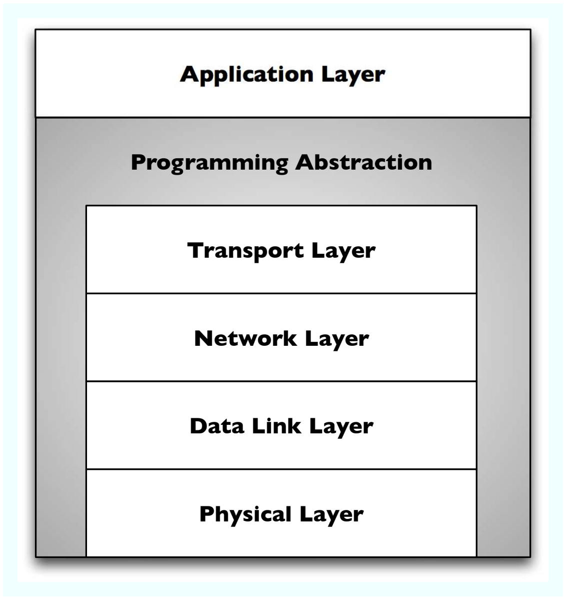

Figure 2.5: Programming models on the WSN protocol stack (adapted from [22])

mechanisms, and placed the techniques and concepts used during the course of this work within this classification.

Chapter 3

Background

This chapter presents a brief discussion of the concepts and algorithms that were used during the course of this work.

This chapter begins with presentation of abstract data types and discusses its applicability for WSNs. The chapter further presents distributed abstract data types (DADTs) and concepts underlying them [24].

This is followed by a brief presentation of the Logical Neighborhoods (LN) [21], a mechanism that enables routing and scoping in WSNs.

3.1 Abstract Data Types

An Abstract Data Type (ADT) is the depiction of a model that presents an abstract view to the problem at hand. This model of a problem typically defines the affected data and the identified operations associated with those.

The set of the data values and associated operations, independent of any specific implementation, is called an ADT [1]. From the application developer's point of view, the use of ADTs allows for the separation of interfaces from specific implementations.

One may consider a *stack* as a simple example of an ADT [13]. It can be represented through the stacked data, and a set of defined operations that include *push(data)*, *pop()*, and *top()*. It is intuitively clear that several different implementations of an ADT may be defined using the proposed specification.

The concept of ADTs has been successfully used in several different areas of science. This idea clearly has found its applicability in WSNs, due to possibility to separate interfaces provided by sensor nodes and their particular implementations.

A top-down aproach can be used to overview an abstraction of WSN and its parts. Any WSN as described earlier, usually consists of a number of sensor nodes, where each sensor node may include several sensors used to measure specific phenomenon.

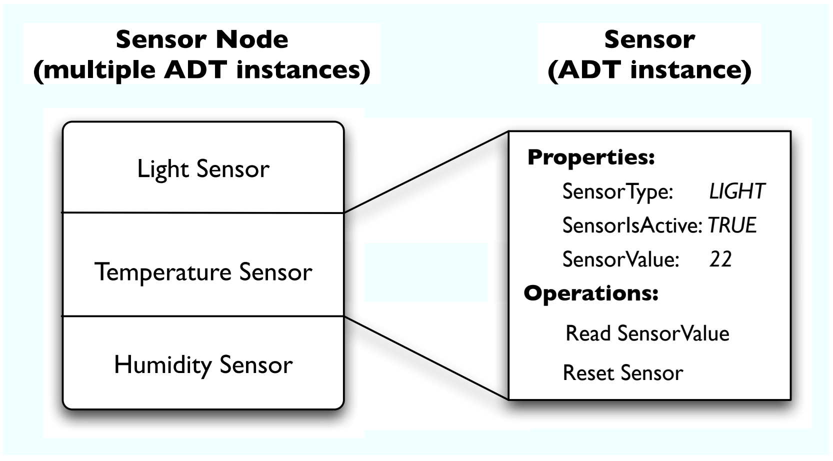

Figure 3.1: Abstraction of sensor node through multiple ADTs

Referring to the example in Figure 3.1, the ADT instance *Sensor* can be used to abstract different types of sensors that may be present on a sensor node. It specifies that a *Sensor* provides the list of common properties and operations. By declaration of multiple such ADT instances, the nature of the wireless sensor node can be abstracted, as can be seen in the *Sensor Node* entity shown in Figure 3.1. This then allows the ADT instances to be used by the application developer at a later point[1].

[1]Further details of ADT specification and instantiation are provided in Section 3.2.4

3.2 Distributed Abstract Data Types

Distributed Abstract Data Types present a new programming language construct used to support distributed and context-aware applications. The concept of DADTs was introduced in [24], and based on the notion of ADTs, which are divided into two classes - data and space ADTs. The rest of this section presents the concepts and provides the reader with the relevant background and examples[2].

3.2.1 Data and Space ADTs

It is important to distinguish between data required by an application, and the location, or space, where the providers of data reside. ADTs in WSNs can provide not only the data from the sensor node, but also express a notion of the "computational environment" hosting the data ADT.

Thus, ADTs be of two types:

- *Data ADTs*
- *Space ADTs*

Data ADTs are "conventional" ADTs which encode application logic, such as, for instance, allowing access to sensor data.

Space ADTs, also known as *sites*, are ADTs that provide an abstraction of the computational environment (in the case of a WSN, a sensor node) that "hosts the data ADT" [24]. The space ADT may use different notions of space, such as physical location or network topology, depending on application requirements as determined by the programmer.

3.2.2 DADTs as an extension of ADTs

Distributed ADTs (DADTs) are an extension of ADTs that make the state of multiple ADTs in a distributed system collectively available. [24].

[2]The examples provided are adapted from [24]

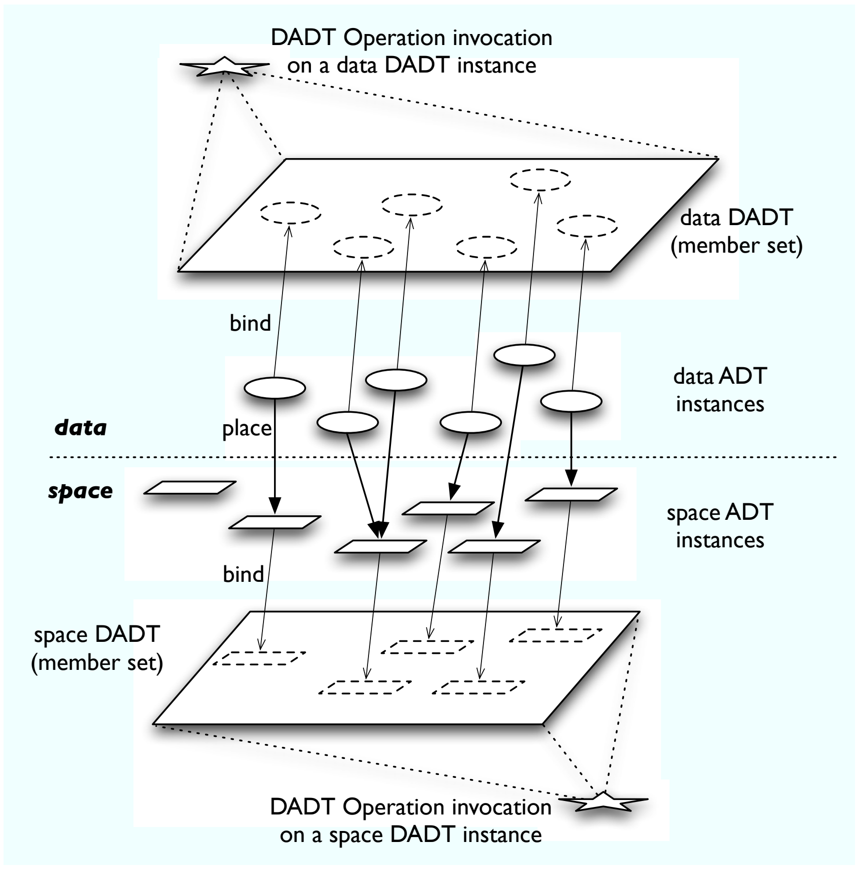

Figure 3.2: Data and space in the DADT model (reproduced from [24])

Similar to ADTs, DADTs provide specifications for distributed data, distributed operators, and constraints. The notion of space is extended to DADTs, and therefore DADTs can either be (as it is shown in the Figure 3.2):

- *Data DADT* that provide distributed access to a collection of data ADTs.
- *Space DADT* that allows distributed access to a collection of space ADTs.

The set of ADTs that are available for collective access using a DADT is called the *member set* of the DADT.

3.2.2.1 DADT Operators and Actions

Operators are used in DADTs to declare distributed references to the ADTs in the DADT member set. This reference conceals identities of individual ADT instances from the application programmer.

DADT operators may belong to one of the following types:

- *Selection Operators* that allow the performance of distributed operations on a subset of the instances in the member set. Examples of such operators could be *all*, or *any* selection operator, because they allow to declare a subset of the instances.
- *Conditional Operators* that provide support for global conditions to be applied on the member set prior to the execution of the DADT operation. One of the examples of conditional operator is operator *in*, that allows to check if one set of instances contains in another one.
- *Iteration Operators* that allow iterating over the ADT instances in the set, and thus permit access to individual ADT instances. Examples of such operators are *next*, *previous*, *last*, *first*, as they allow to address and iterate over the members of the set.

The DADT operators described above allow different way to access ADT instance and execute one of the ADT operations defined in the ADT specification.

Distributed appliation developer may require to implement more complicated application logic, though it may happen that even available ADT operations are not sufficient. In this case the special DADT construct, called DADT action, can be used [24].

DADT actions are specified by the DADT type, but are executed similarly to ADT operation on remote ADT instances.

3.2.2.2 Views

DADT Views permit the definition of the scope of distributed operations that the application requires to perform. This approach is particularly useful when a distributed operation has to be executed only on a subset of the member set of ADT instances. The concept of DADT Views is presented in Figure 3.3.

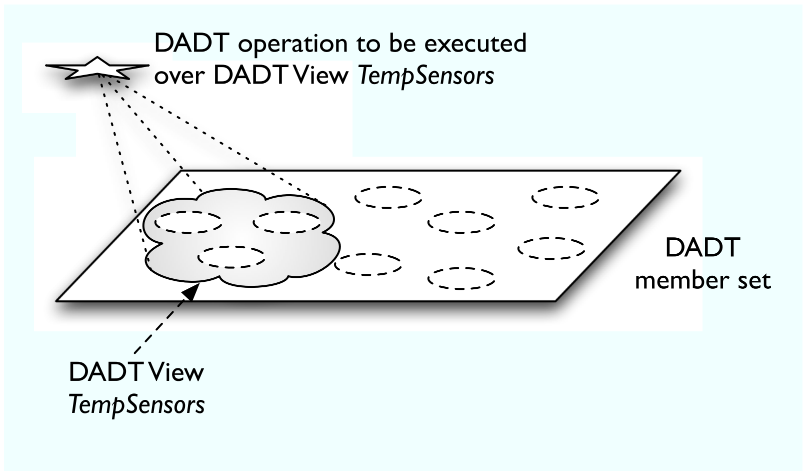

Figure 3.3: DADT views (reproduced from [24])

The member set may be partitioned into DADT Views by using properties. A *Property* is a DADT characteristic that is defined in terms of an ADT's data and operations, and is evaluated locally on the ADT instance [24]. DADT View may either be:

- *Data View*,
- *Space View*.

Application developer should decide what could be the best way to access the WSN. A Data DADT instance can be used to operate on the distributed data and limit the scope using predicates over DADT properties representing data, space, or both, depending on the needs of the application. Similarly, any kind of view to access the distributed representation of Space can be applied.

This section has presented concepts underlying the DADTs. A brief description of the DADT prototype [24] will be presented further in Section 3.2.3.

3.2.3 The DADT Prototype

The DADT prototype implemented by Migliavacca et al [24] was written in Java, and presents the design of a DADT specification language that extends Java.

This section provides the reader with further details on the concepts central to DADTs[3], a description of the existing DADT prototype [24], and a discussion on its limitations.

3.2.4 ADTs Specification and Instantiation

As was mentioned in Section **??**, each sensor node in a WSN can be abstracted using multiple ADT instances (see Figure 3.1), and therefore conceal the details of sensor node abstraction from the application developer. The code snippets below are provided to illustrate these concepts.

The specification of a described ADT may be defined as a Java class, as shown in Listing 3.1

```
class Sensor {
  //data properties of the sensor
    int sensorType;
    double sensorReading;
    boolean active;
  //operations that can be performed on the sensor
    public double read(){ //read the sensor value.
    ...
    }
    public void reset(){ //reset the sensor
    ...
    }
```

Listing 3.1: Sensor ADT instances

This specification declares that a Sensor ADT instance should provide the following properties:

[3]All the DADT concepts presented below are described using the DADT specification language.

- an integer value to define the sensor type,
- a double value that holds the sensor data,
- a boolean value that stores information about sensor's state of activity.

as well as operations that allow to *Read* sensor data and *Reset* the sensor.

It is possible to define multiple such instances of this specification. For example, the ADT instances for a given sensor node that has two kinds of sensors - (a) a temperature sensor, and (b) a light sensor - may be defined using the ADT specification described above, as shown in Listing 3.2.

```
  // Temperature sensor ADT instance
    Sensor temperatureSensor = new Sensor(TEMPERATURE);
  // Light sensor ADT instance
    Sensor lightSensor = new Sensor(LIGHT);
```

Listing 3.2: Sensor ADT instances

3.2.5 DADT Specification and Instantiation

The concept of DADTs was first introduced in Section 3.2, and will be extended in this chapter with examples of DADT specification and instantiation.

DADT specifications can be best understood by carrying forward the example described in Section 3.2.4. To allow for collective access to multiple ADT instances of the type specified in Listing 3.1, a DADT *DSensor* may be defined as shown in Listing 3.3.

```
class DSensor distributes Sensor{
  //properties:
    property isSensorType(int type);
    property isActive();

  // distributed operations:
    distributed double average()
    distributed void resetAll();
}
```

Listing 3.3: Data DADT specification (reproduced from [24])

This DADT specification allows two simple distributed operations to be performed on multiple data ADTs of type *Sensor*:

- *resetAll()*
- *average()*

The *resetAll()* operation is used to reset every sensor in the DADT member set, or subset of ADT instances defined by a DADT View (see Section 3.2.2.2).

The *average()* operation allows for the calculation of the average of the readings of every sensor in the member set, or the subset of it defined by the DADT view.

Additionally, the DADT specification shown in Listing 3.3 declares two DADT Properties described later.

Listing 3.4 shows how DADT specifications can be instantiated as an object of a Java class, and be used to perform defined distributed operations.

```
DSensor ds = new DSensor();
ds.resetAll();
```

Listing 3.4: DADT Instantiation (reproduced from [24])

3.2.6 Binding ADTs to DADTs

As mentioned earlier in Section 3.2.2, a DADT member set consists of the set of ADTs that are available for collective access, which, in the given example, is the collection of ADTs of type *Sensor*.

An ADT instance is made part of the member set by binding it to the DADT type. This can be done using a dedicated programming construct *bind* as shown in Listing 3.5, where the Sensor ADT defined in Listing 3.1 is bound to the DADT type *DSensor* defined in Listing 3.3.

```
  bind(new Sensor(TEMPERATURE), "DSensor");
  bind(new Sensor(LIGHT), "DSensor");
  ...
```

Listing 3.5: Binding ADT instances to a DADT instance

Binding of ADT instances takes place on the sensor device that holds sensors, as those are abstracted into specific ADTs.

3.2.7 Implementing DADT Operators and Actions

The concepts of DADT Operators and Actions were already introduced in the previous chapter (see Section 3.2.2.1). This section describes the implementation of Operators and Actions in the DADT prototype [24].

3.2.7.1 DADT Operators

Listing 3.6 extends the example first introduced in Listing 3.3, and presents the use of the DADT selection operator *all*.

```
class DSensor distributes Sensor{
  ...
  distributed void resetAll(){
    (all in targetset).reset();
  }
}
```

Listing 3.6: Use of DADT Selection Operator

The distributed operation *resetAll* uses the selection operator *all* in order to gain access to all ADT instances in the DADT target set, and subsequently invokes the *reset* operation on the ADT instance. The *reset* operation was declared in the ADT specification shown in Listing 3.1.

3.2.7.2 DADT Actions

However, if the application developers finds the set of available ADT operations limited, he/she has the option of using DADT actions that are defined in the DADT type and executed locally on the ADT instance.

DADT actions do not necessarily consist only of single ADT operations, but can also implement more complicated logic. For instance, a *reliableRead* action may be implemented that is capable of handling sensor read failures by means of performing multiple read attempts, failing which a sensor node reset is performed (See Listing 3.7).

```
class DSensor distributes Sensor {
  distributed double average(){
  ...
  action double reliableRead(){
    double reading;
      int tries = 3;
      while (tries > 0){
        reading = local.read();  // use of ADT operation
        if (reading == ERROR) --tries;
        else break;
      }
      if (reading == ERROR) {
        local.reset();
        reading = local.read();
      }
    }

    double[] sensorReadings = (all in targetset).reliableRead();
    ...
    // evaluation of the average value based on received readings
    ...
    }
}
```

Listing 3.7: Use of DADT Action (reproduced from [24])

3.2.8 DADT Views

DADT Views are an effective tool for the application developer to define the scope of a distributed operation. As was described in the Section 3.2.2.2, DADT Views are created using DADT properties.

To continue to use the example running throughout this section, if the application programmer wishes to refer to a subset of temperature sensors from among the member set of ADT instances bound to the DADT type *DSensor*, a data view *TempSensors*, as shown in Listing 3.8, can be declared.

```
dataview TempSensors on DSensor as isSensorType(TEMPERATURE) &&
   isActive();
```

Listing 3.8: Definition of DADT Data View

The DADT name *DSensor* in this case refers to its member set, and the data view *TempSensors* is defined as a subset of this member set and consists only of sensor nodes with temperature sensors for which the evaluation of both DADT properties (See Listing 3.9) *isActive* and *isSensorType* return *true*.

```
class DSensor distributes Sensor {
  property isSensorType(){
         return (local.type == type);
  }
  property isActive() {
        return local.isActive();
  }
  ...
}
```

Listing 3.9: Definition of DADT Properties

This section has described the DADT prototype presented in [24], which was modified and extended to use in WSNs. Chapter 5 provides further details about limitations of the given prototype and presents the DADT/LN prototype.

3.3 Logical Neighborhoods

Typically, communication between WSN nodes is based on routing information between nodes by exploiting the communication radius of each node. The notion of a node's physical neighbourhood - the set of nodes in the network that fall within the communication range of a given node - is central to a mechanism of this nature.

However, in heterogenous WSN applications, the developer might require to communicate with a specific subset of the network that is defined logically and not physically. As an example of this, consider the following case. An application that provides security in a high-risk environment by monitoring motion might require - in the event of a security alarm - all sensors at the entrances to the guarded area to report about detected motion. The sensors at the entrance form a logical neighbourhood in this case. However, as the entrances may be widely separated, it is not for granted that these nodes constitute part of a single physical neighbourhood.

The use of current WSN programming techniques to enable a mechanism of this nature entails additional programming effort, because the developer has to deal not only with the application logic, but also with the underlying problems of transmitting messages to a specific logical neighbouhood while using lower layer constructs that have no notion of this. This leads to increased code complexity [20].

Mottola and Picco [20] suggest the addressing of the aforementioned issues using *Logical Neighbourhoods (LNs)*, an abstraction that replaces the node's physical neighbourhood with a logical notion of proximity.

Figure 3.4 provides comparison between physical and logical neighbourhoods. The black node represents the node that defines the sensor node with its neighbourhoods. The light grey nodes connected by the dashed circle represent the physical neighborhood, whereas the dark grey ones with the solid polygon represent (one of the nodes in) the node's logical neighborhood.

Using this abstraction, programmers can communicate with members of a LN using a simple message passing API, thereby allowing for logical broadcasts. The

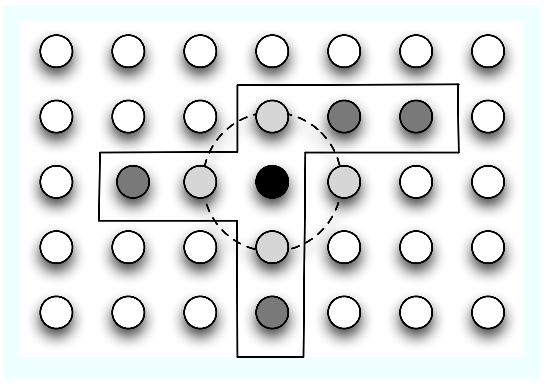

Figure 3.4: Representation of logical and physical neighborhood of a given node. (reproduced from [20])

implementation of this API is supported by means of a novel routing mechanism devised specifically to support LN communication.

3.3.1 The LN Abstraction

LNs can be specified using a declarative language such as SPIDEY [20, 21], and involves the definition and instantiation of the *node* and the *neighbourhood*.

Nodes are a logical representation of the subset of a sensor node's state and characteristics, and are used for the specification of an LN. Nodes are defined in a node template and are subsequently instantiated, as shown in Listing 3.10

A neighbourhood can be defined by applying predicates on the attributes defined in the node template. The neighbourhood is defined using a neighbourhood template, and subsequently instantiated.

```
node template Sensor
  static Function
  static Type
  dynamic BatteryPower
  dynamic Reading

create node ts from Sensor
  Function as "Sensor"
  Type as "Temperature"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

Listing 3.10: Node Definition and Instantiation

Listing 3.11 shows the definition and instantiation of a neighbourhood - based on the node template defined in Listing 3.10 - which selects all temperature sensors where the reading exceeds a threshold.

```
neighbourhood template HighTempSensors(threshold)
  with Function = "Sensor"
      and Type  = "Temperature"
      and Reading > Threshold

create neighbourhood HigherTemperatureSensors
  from HighTempSensors(threshold:45)
```

Listing 3.11: Neighbourhood Definition and Instantiation

LN communication is enabled using a simple API that overrides the traditionally used broadcast facility and makes it dependent on the (logical) neighbourhood the message is addressed to [20]. The underlying routing mechanism is presented in further details in [20, 21].

3.4 Summary

This chapter presented an overview of the concepts, technologies, and hardware underlying the work presented in this thesis. After underlining the need for the use of programming models to increase the ubiquity of WSN use, this chapter introduced a taxonomy of programming abstractions and placed the Distributed Abstract Data Types programming abstraction used in this work within the framework of the taxonomy. This was followed by a detailed discussion on Abstract and Distributed Abstract Data Types used in the application layer of the prototype produced during the course of this work, and the Logical Neighbourhood routing mechanism used in the network and data link layers of the prototype.

Chapter 4

Tools

This chapter presents the tools that were used to show the feasibility of the prototype implemented during the course og this work. These tools include JiST/SWANS discrete event simulator and Sun Small Programable Object Technology (Sun SPOT) hardware platform.

4.1 JiST/SWANS simulator

As the simulator used in this work is a discrete event simulator, this section begins with a short description of discrete event simulators. This is followed by a discussion on the SWANS network simulator used during the course of this work, and the JiST Java-based discrete event simulator upon which SWANS is built.

4.1.1 Discrete Event Simulator

A discrete event simulator allows for the simulated execution of a process (that may be either deterministic or stochastic), and consists of the following components [25]:

- *Simulation variables:* These variables keep track of simulation time, the list of events to be simulated, the (evolving) system state, and performance indicators.

- *Event handler:* The event handler schedules events for execution at specific points in simulation time (and unschedules them if necessary), and additionally updates the state variables and performance indicators.

4.1.2 Java In Simulation Time (JiST)

JiST [7] is a discrete event simulator that is efficient (compared to existing simulation systems), transparent (simulations are automatically translated to run with the simulation time semantics), and standard (simulations use a conventional programming language, i.e., Java).

JiST simulation code is written in Java, and converted to run over the JiST simulation kernel using a bytecode-level rewriter[1], as can be seen in Figure 4.1.

The execution of a JiST program can be understood by considering the example shown in Listing 4.1

```
import jist.runtime.JistAPI;
class hello implements JistAPI.Entity {
  public static void main(String[] args) {
    System.out.println("Simulation start");
    hello h = new hello();
    h.myEvent();
  }
  public void myEvent() {
    JistAPI.sleep(1);
    myEvent();
    System.out.println("hello world, " + JistAPI.getTime());
  }}
```

Listing 4.1: Example JiST program (reproduced from [7]

This program is then compiled and executed in the JiST simulation kernel, using the following commands:

```
javac hello.java
java jist.runtime.Main hello
```

[1]The bytecode rewriter and the simulation kernel are both written in Java

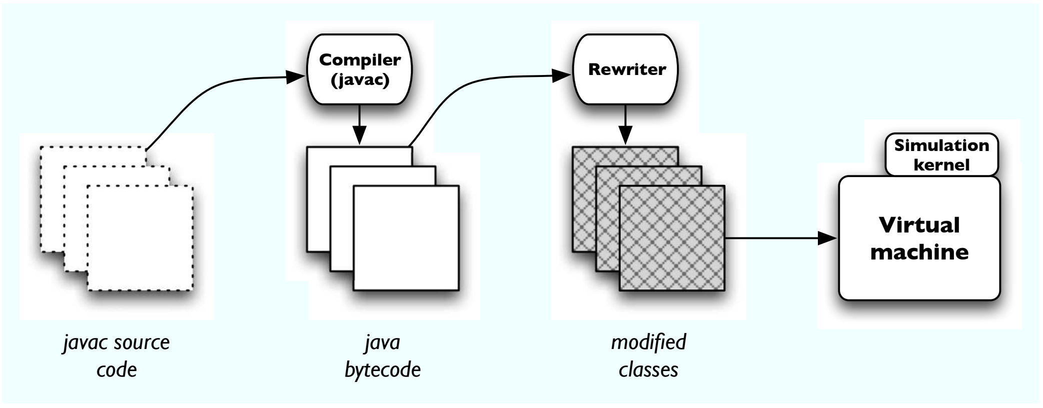

Figure 4.1: The JiST system architecture (reproduced from [7])

Listing 4.2: Execution of the program in the JiST

The simulation kernel is loaded upon execution of this command. This kernel installs into the JVM a class loader that performs the rewrite of the bytecode. The JiST API functions used in the example code are used to perform the code transformations. The method call to myEvent is now scheduled and executed by the simulator in simulation time. Simulation time differs from “actual” time in that the advancement of actual time is independent of application execution, whereas simulation time is not.

4.1.3 Scalable Wireless Ad hoc Network Simulator (SWANS)

SWANS is a wireless network simulator developed in order to provide efficient and scalable simulations without compromising on simulation detail [6], and is built upon the JiST discrete event simulator described in Section 4.1.2. It is organised a a collection of independent, relatively simple, event driven components that are encapsulated as JiST entities.

SWANS has the following capabilities [6]:

- The use of interchangeable components enables the construction of a protocol stack for the network, and facilitates parallelism, and execution in a

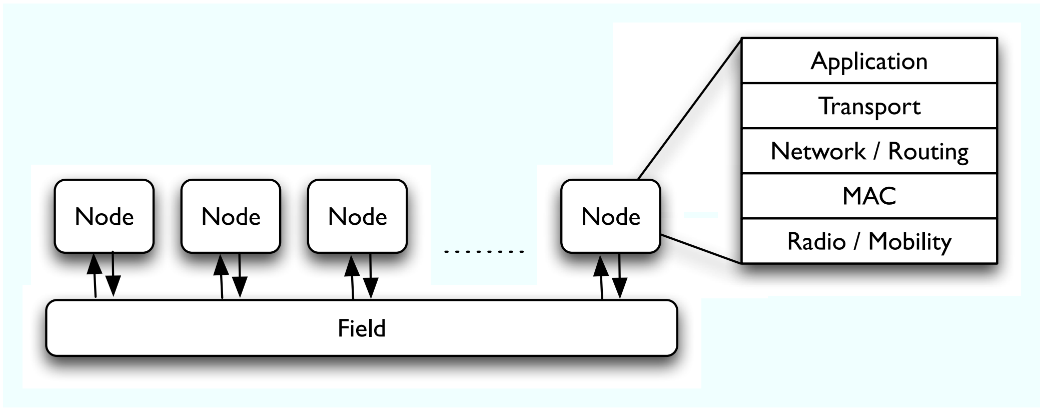

Figure 4.2: SWANS architecture

distributed environment.

- Can execute unmodified Java network applications on the simulated network (in simulation time), by virtue of its being built over JiST. Using a harness, the aforementioned Java code is automatically rewritten to run on the simulated network.

The SWANS architecture may be seen in Figure 4.2.

4.2 Sun SPOT platform

Sensor nodes, as mentioned in Section 2.1, are characterised by limited resources, including memory.

In order to overcome memory limitations, wireless sensor network applications have traditionally been coded in non-managed languages like C and assembly language [26].

Managed runtime languages like Java were not used for sensor network programming because of the combination of the static memory footprint of the Java Virtual Machine (JVM) and the dynamic memory footprint of the WSN application code.

On the other hand, it is widely accepted that development times are greatly reduced upon the use of managed runtime languages such as Java [26]. Therefore, currently prevalent WSN programming practice trades developer efficiency for memory efficiency.

However, Simon et al [26] state the benefits resulted from using a managed runtime language for WSN programming as follows:

- Simplification of the process of WSN programming, that would cause an increase in developer adoption rates and productivity.
- Opportunity to use standard development and debugging tools.

The use of the Java programming language in SunSPOTs makes it particularly suitable as a platform for the DADT applications presented. This is because the DADT programming abstraction is designed to reduce programmer workload, and the use of a managed runtime language such as Java has been shown to further improve developer efficiency.

Sun Microsystems has, on the basis of the arguments discussed in the previous section, proposed and built a sensor device called the Sun Small Programmable Object Technology (Sun SPOT) that uses a on-board JVM to allow for WSN programming using Java.

The Sun SPOT (see Figure 4.3) uses an ARM-9 processor, has 512 KB of RAM and 4 MB of flash memory, uses a 2.4GHz radio with an integrated antenna on the board. The radio is a TI CC2420 and is IEEE 802.15.4 compliant.

4.2.1 The Squawk JVM

The Squawk JVM is used on Sun SPOTs to enable on-board execution of Java programs. The Squawk VM was originally developed for a smart card system with even greater memory constraints than the Sun SPOTs. The Squawk JVM has the following features [26]:

Figure 4.3: Sun SPOT device

- It is written in Java, and specifically designed for resource constrained devices, meeting the requirements of Connected Limited Device Configuration 1.1 (CLDC) framework for Java Micro Edition (Java ME) applications.
- It does not require an underlying OS as it runs directly on the Sun SPOT hardware. This allows for a reduction in memory consumption.
- It suuports inter-device application migration.
- It allows the execution of multiple applications on one VM, representing each one as an object.

As resource constrained devices are incapable of loading class files on-device by virtue of their limited memory, a VM architecture known as the “split VM architecture” is used, as shown in Figure 4.4.

The Squawk split VM architecture uses a class file preprocessor known as the *suite creator* that converts the *.class* bytecode into a more compact representation called the Squawk bytecode. According to [26], Squawk bytecodes are optimised in order to:

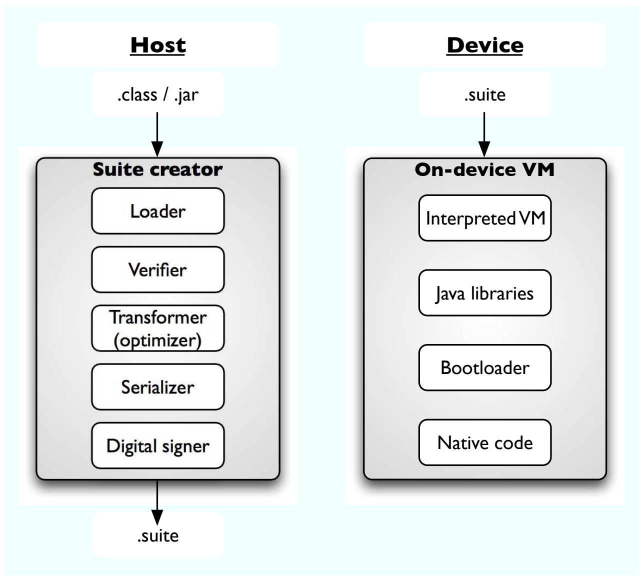

Figure 4.4: The Squawk Split VM Architecture (reproduced from [26])

- minimise space used by using smaller bytecode representation, escape mechanisms for float and double instructions, and widened operands.
- enable in-place execution, by “resolving symbolic references to other classes, data members, and member functions into direct pointers, object offsets and method table offsets respectively”.
- simplify garbage collection, by the careful reallocation of local variables, and by storing on the operand stack the operands of only those instructions that would result in a memory allocation.

The Squawk bytecodes are converted into a *.suite* file created by serialising and saving into a file the internal object memory representation. These files are loaded on to the device, and subsequently interpreted by the on-device VM.

4.2.2 Sun SPOT applications

Sun SPOT applications are divided into two classes: [27]:

- *On-SPOT applications*

- *On-Host applications*

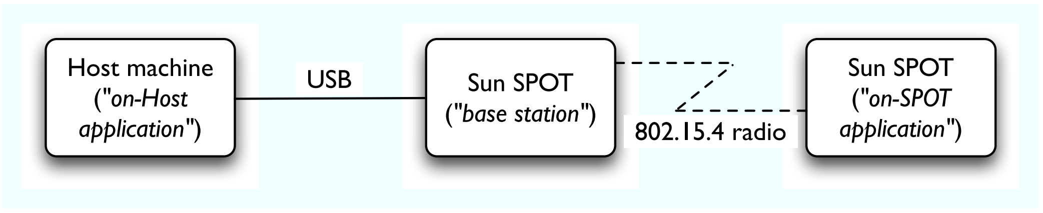

Figure 4.5: Types of Sun SPOT applications (adapted from [27])

On-SPOT applications are deployed and executed on a remote Sun SPOT that communicates untethered. On-SPOT applications are Java programs that runs on the Squawk VM, and are compliant with CLDC 1.1 specification.

On-Host applications run on the host machine (typically a PC), and communicate with the network of Sun SPOTs through a base station node that serves no purpose other than to facilitate Sun SPOT-host machine communication.

The base station node, which is a Sun SPOT itself, communicates with other nodes in the network using RF communication, and with the host machine via a USB link (see Figure 4.5). The host application is a Java 2 Standard Edition (J2SE) program.

4.3 Summary

This chapter presented tools that were used to demonstrate the feasibility of the prototype implemented as part of this work. The chapter provided an overview of the JIST/SWANS network simulator used for verification of the prototype. This was followed by a presentation of the SunSPOTs hardware platform, and its particular suitability for the application under consideration.

Chapter 5

Design Overview

This chapter presents the details of the DADT/LN prototype implemented as part of this work. The chapter begins with an outline of the limitations of the existing DADT prototype [24]. This is followed by a discussion on the architecture of the hybrid DADT/LN prototype that combines the functionality of the DADT prototype with the LN routing mechanism, and extends the existing prototype to work on WSNs.

5.1 Limitations of the existing DADT Prototype

The DADT prototype proposed in [24] enables the use of DADTs to facilitate distributed application programming.

It supports Java-based application development, and consists of two parts:

- *Translator*
- *Runtime library*

The *translator* translates Java programs extended with DADT programming constructs into conventional Java classes.

The *runtime library* is used during the translation stage, and allows for the execution of the Java classes in the J2SE. It provides support for DADT constructions and methods, such as *binding* ADTs to DADTs, *DADT Views*, *Actions* and *Operators*.

The communication in the prototype uses IP Multicast, and allows the delivery of information to ADTs bound to a specific DADT[1].

The DADT prototype is a proof of the DADT concept. While this approach is clearly applicable to WSNs, the prototype itself did not support WSN abstractions, as it suffers from the following major limitations:

- The lack of a routing mechanism suited for WSN
- Limitations in portability to real WSN nodes
- Memory and code occupation

5.2 The DADT/LN Prototype Design

The DADT prototype proposed in [24] proved that the concept of DADTs could possibly be applied to WSN applications, but existing limitations in the prototype prevent its use in WSN simulators or real nodes.

This work makes the following contributions:

- Enhances the DADT prototype for use in WSNs by extending it to run on simulators as well as devices in a real-world environment.
- Interfacing the LN mechanism presented in [19] with the DADT prototype in order to enable abstracted communication between groups of nodes in the WSN defined by DADTs.
- Verifies the utility of DADT abstractions in the WSN application layer.

5.2.1 Workflow Overview

The overview of the workflow involved in using DADTs to enable WSN application programming is shown in Figure 5.1. The application developer writes the

[1]Group of ADTs of this kind are defined as multicast groups in the prototype.

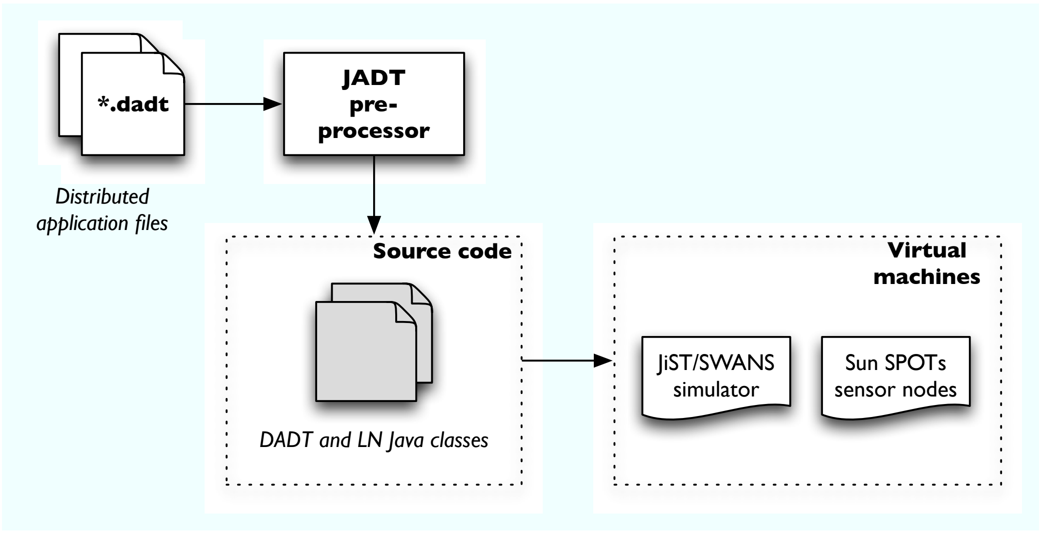

Figure 5.1: Workflow for development of an application that uses the DADT/LN prototype

application layer code for the WSN in a series of *.dadt* files using the DADT specification language. The JADT preprocessor is then used to convert the code written by the application programmer into Java code that interfaces with the DADT infrastructure (extended from the prototype presented in [24]). In order to facilitate routing using LNs, the DADT infrastructure is interfaced with a previously developed implementation of LNs [19].

The application (including the implementation of layers lower in the protocol stack) is then loaded on to either:

- the JiST/SWANS simulator [7, 6] (See Section 4.1 for the implementation details of the simulator)
- a collection of Sun SPOT wireless sensor devices [26] (see Section 4.2) to execute the given application on real sensor nodes.

The WSN in the DADT/LN prototype developed as part of this work consists of two types of devices each of which conceal a number of objects on the different layers of the WSN stack, as shown in Figure 5.2:

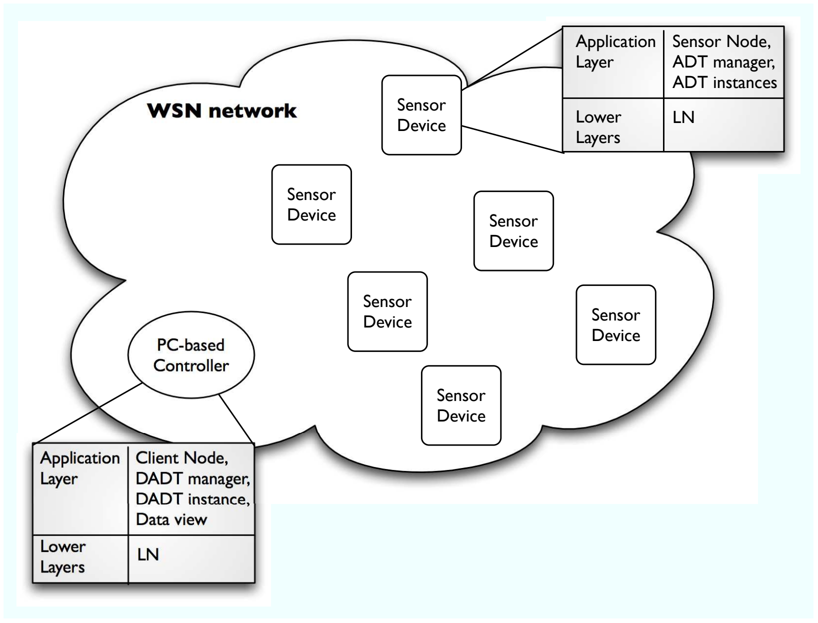

Figure 5.2: Schematic representation of the WSN as abstracted in the DADT/LN prototype

- *DADT Controller Device*
- *Sensor Device*

*T*he DADT Controller Device can reside, depending on the application, either on a real-world sensor device such as a Sun SPOT, or be a PC-based application. On the application layer, this abstraction includes the distributed application code, the DADT instance, and the DADT manager. The network layer entity hosts the LN implementation.

*T*he Sensor Device is a real-world sensor device such as a Sun SPOT, which holds the application layer entities, which includes a Sensor Node abstraction consisting of multiple sensor ADT instances, and an ADT manager, and a network layer entity represented by the LN implementation.

5.2.2 Architecture

The architecture of the DADT/LN prototype is presented in Figure 5.3, and consists of the following logical layers.

5.2.2.1 Upper layer

The upper layer consists of the following components:

- *Distributed application*
- *Sensor* and *DSensor*
- *DADT runtime layer*
- *DADT manager*
- *The Property, Action, and View classes*

The Distributed application comprises the code implemented by the application developer, and is written in the DADT specification language. This code is later translated into executable Java code using the JADT preprocessor, and interfaced with the DADT runtime (see Section 5.2.1).

The *Sensor* and *DSensor* classes are the data ADT and data DADT specifications used by the DADT/LN prototype, and are entirely defined by the application developer. The space ADTs *Host* and *Network* are currently built into the DADT/LN prototype, and cannot be defined by the application programmer.

The *DADT runtime layer* holds the DADT runtime library, which provides handling of ADTs and DADTs.

The *DADT Manager* performs several tasks, including managing the binding and unbinding of ADT instances to DADT type by means of the *Binding Registry* class, and providing support for space ADTs using the *Site* class.

The abstract classes *Property* and *Action* represent the corresponding concepts described in Section 3.2.2. The class *View* provides support for data and space

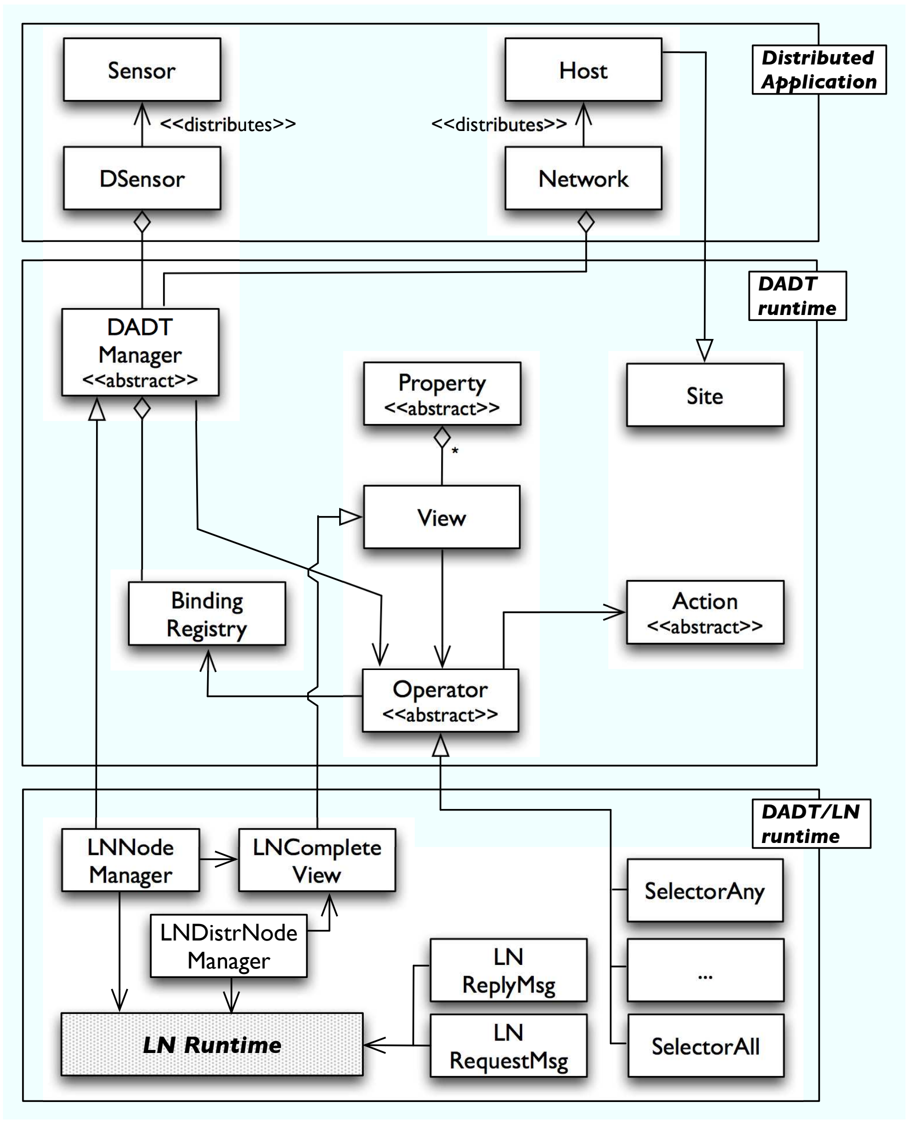

Figure 5.3: Architecture of the DADT/LN prototype

DADT Views, and contains the set of *Property* objects that define the scope of the operations.

Properties are organised into an abstract tree that codifies the logical predicates defining the view, with the leaves representing DADT properties and the nodes specifying boolean operators that are used to compose it. The class *Operator* is a superclass for all of the DADT Operators, though the current implementation of the DADT/LN prototype provides support mainly for selection operators.

5.2.2.2 Lower Layer

The *DADT/LN runtime* layer provides support for interfacing the DADT runtime with the LN runtime [2], which enables routing and scoping mechanisms in WSN. Interaction with LN runtime achieved by using instances of the *LNNodeManager* class on each sensor device, and the *LNDistrNodeManager* instance on the controller node.

Communication between sensor devices and the controller node is fully handled by the LN runtime layer. This is abstracted from the application programmer by means of the *Request* and *Reply* messages (details of implementation are provided in Section 6.3). The classes *SelectAll* and *SelectAny* implement DADT selection operators, and allow for the invocation of the corresponding send methods defined in the LN API.

5.3 Summary

This chapter presented the architecture of the DADT/LN prototype developed during the course of this work. The chapter concluded with a detailed description of the modules constituting the prototype and interaction between them.

[2]The LN implementation used as part of this work does not use a declarative language for LN specification which was used for decription of the LN concepts

Chapter 6

Implementation and Testing

This chapter provides the implementation details of the DADT/LN prototype developed as part of this work.

6.1 The DADT/LN Prototype on the Controller

Figure 6.1 presents the operation of the DADT/LN prototype on the controller. Depending on application requirements, the Controller can either be run on a separate node in WSN, or be a PC-based application.

As shown in the figure, the implementation running on the Controller consists of the following entities:

- *Client Node*
- *DADT Instance*
- *Expression Tree*
- *DistrNode Manager*
- *Data View*

The *Client Node* is an abstraction that holds a DADT instance. The application programmer's requests to the network are issued by the Client Node.

The *DADT Instance* allows for collective access to multiple ADT instances (see Section 3.2.5).

The *Expression Tree* is a special object that allows the construction of an abstract tree for the DADT View, based on the set of DADT Properties defined.

A *DistrNode Manager* provides the interface between the Client Node and the network, and passes request messages from the Client Node to the lower layers of the protocol stack.

DADT Views present a mechanism for partitioning the collection of ADT instances bound to a particular DADT type, and were explained in Section 3.2.2.2.

6.1.1 Workflow

The instantiation of a DADT type by the application programmer's code causes the following actions to take place at the Controller:

- The Client Node creates an instance of the DADT type and uses it to perform collective operations on the network.
- A new expression tree object is created to provide a representation of the DADT View defined by the application programmer.
- The DADT instance creates an instance of the DistrNode Manager.

When the application programmer's code requests the execution of a distributed operation, the Client Node forwards the request to the DADT instance, which in turn processes the request and facilitates communication across the WSN.

The DADT instance performs the following actions:

- It processes the request, and, according to the defined scope of the distributed operation, constructs the relevant DADT View using the expression tree object.
- It uses the DistrNode Manager to translate the DADT View (defined in the expression tree object) into LN predicates, and constructs and sends a request message to the underlying WSN.

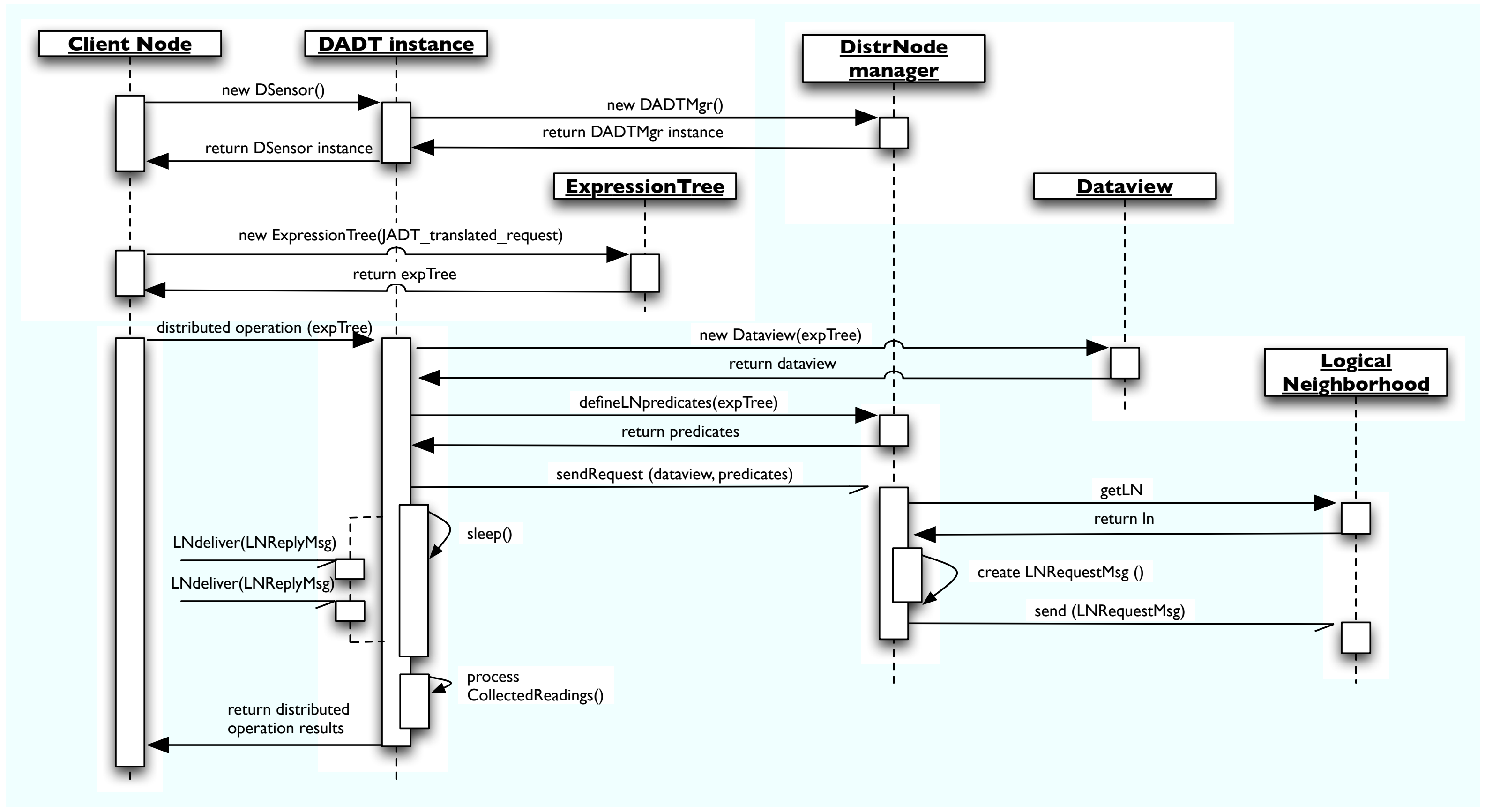

Figure 6.1: Operation of the DADT/LN prototype on Controller

- The DADT instance sleeps until, if required, the result of the request is received from the network layer.

If no reply is expected from the nodes to which message was sent, the Client Node continues to operate normally. This situation occurs, for instance, in the case of Controller Node requiring each sensor node to execute the *ResetAll* action.

Otherwise, if the result of the distributed computation is expected, the DADT instance is notified when the relevant data are received from the WSN. The DADT instance then invokes the relevant methods to collect and process the received readings. Finally, it returns the result of the DADT request to the Client Node.

6.1.2 Interaction between DADT and LN layers

The DistrNode Manager is responsible for the interaction between DADT and LN layers, as was mentioned in Section 5.2.2.2.

Invocation of the request for distributed operation causes the DistrNode Manager to perform the following actions:

- Provide mapping between components of the Expression Tree object, which represents the scope of the distributed request and LN predicates.

- Identify the required selection operator provided in the request (See Section 3.2.2.1 for details) and send this request to the WSN.

Each expresson tree contains a set of DADT Properties, comparison operators and values associated with each DADT Property. Moreover, each DADT Property implements method that provides mapping of the DADT Property into LN predicate by using the relevant LN runtime classes.

Therefore, when request has to be sent, the DistrNode Manager performs traversal of the given Expression Tree object and collects a list of LN predicates that is used later by LN runtime for routing. Listing **??** presents an example of such a mapping.

```
public class DSensor_isValueLess_Property implements  Property {

  private int valueToCheck;
  private int type;

  public DSensor_isValueLess_Property(int value, int type)  {
    this.valueToCheck = value;
    this.type = type;
  }
  ...
  public Predicate getLNPredicate() {
    return(new IntegerSimplePredicate("value_" + type,
        IntegerSimplePredicate.LESS_THAN, this.valueToCheck));
  }

  public boolean evaluate(Object o) {
    Sensor local = (Sensor) o;
    if (local.type == type){
      return (local.getSensorReading() < valueToCheck);
    }
    else
      return false;
  }
}
```

Listing 6.1: Mapping DADT Property to LN predicate label

DADT Property shown in Listing **??** checks if the sensor of a given type provides a sensor reading which is less than specified by *valueToCheck* field. When the request is constructed, method *getLNPredicate* is used and mapping is performed.

Currenty the mapping between each DADT Property and relevant LN Predicate object has to be provided by an application developer manually. Future work may include implementation of such mapping in automatic mode.

6.2 The DADT/LN Prototype on the Sensor Device

In the rest of this section, the term *sensor device* is used to refer to the physical sensor node entity, while the term *sensor node* refers to the application layer abstractions of all of the sensors within the device.

Figure 6.2 presents the operation of the DADT/LN prototype on the sensor device, which may be either a simulated or a Sun SPOT node.

The DADT/LN prototype implementation running on each sensor node consists of the following entities:

- *Sensor Node*
- *Sensor ADT instance*
- *Node Manager*

A *Sensor Node* is an abstraction that consists of a list of sensors. This follows from the example used to illustrate the concept of ADTs in Section 3.2.4.

A *Sensor ADT instance* is the ADT instance of a given sensor on the sensor node, upon which the prototype executes.

A *Node Manager* provides the interface between the sensor node and the network, thereby abstracting sensor ADT instances from queries issued by the DADT instance at the (PC-based) controller.

6.2.1 Workflow

Initialisation of the Sensor ADT instance is performed possibly multiple times on a given Sensor Node, as a node might consist of multiple sensors. Following this, the sensor ADT instances are bound to a particular DADT type by calling the Node manager[1].

When the lower layer (which runs the LN algorithm) delivers a message to the Sensor Node, the Node Manager is used to process the request message. The

[1]The Node manager is assumed in our current implementation to be aware of all DADT types defined in the WSN.

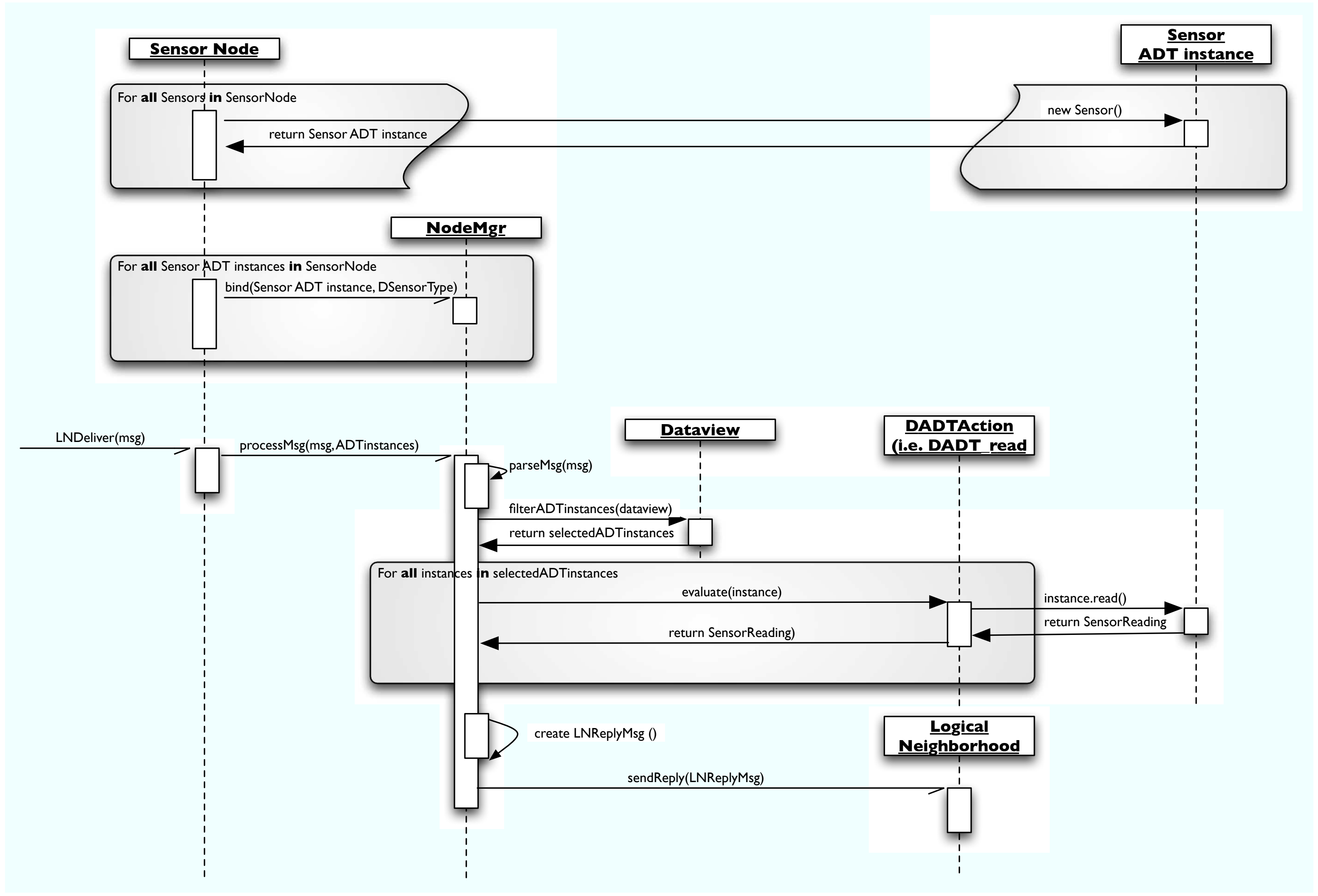
Sensor Node
Sensor ADT instance
For all Sensors in SensorNode
new Sensor()
return Sensor ADT instance
NodeMgr
For all Sensor ADT instances in SensorNode
bind(Sensor ADT instance, DSensorType)
Dataview
DADTAction (i.e. DADT_read
LNDeliver(msg)
processMsg(msg,ADTinstances)
parseMsg(msg)
filterADTinstances(dataview)
return selectedADTinstances
For all instances in selectedADTinstances
evaluate(instance)
instance.read()
return SensorReading
return SensorReading)
Logical Neighborhood
create LNReplyMsg ()
sendReply(LNReplyMsg)

Figure 6.2: Operation of the DADT/LN prototype on sensor device

request message contains details of the DADT View (see Section 3.2.2.2) used later to filter from the sensor ADT instances on the given Sensor node those that fit into the requested DADT View.

The request message also contains information about the DADT action to be performed on device (see Section 3.2.2.1). The Node manager calls the action for each sensor ADT instance that fits into the DADT View.

If the application layer requires that a reply be sent, the LN implementation in the lower layer of the protocol stack is used as can be seen on the bottom right section of Figure 6.2.

6.2.2 Handling of Multiple ADT Instances

When the request message is received in the application layer, then at least one of the sensor ADT instances in the Sensor Node fits into the DADT view[2], because LN runtime uses a set of LN predicates based upon DADT view (See Section 6.1.2) for routing. This minimises the number of messages received at the application layer. However, since a given Sensor Node may contain several sensor ADT instances, the ADT instances have to be filtered.

Filtering is performed by using DADT view object that was sent as the part of Request message. For each ADT instance binded to the DADT class, all DADT properties that constitute the DADT view are evaluated.

Once selection of the relevant for the request ADT instances is done, the request DADT Action is performed by the *Node Manager*.

6.3 Communication Messages

As was discussed in Chapter 5, nodes that use the DADT/LN prototype are communicate between each other by means of specific messages:

- *Request message*

[2]If none of the sensor ADT instances fit the view, the message would have been discarded in the network layer.

- *Reply message*

The *Request Message*, as follows from its name, used by controller device to define a request for distributed action. This message contains the following fields:

- *Sender*, which provides information about the node-requestor, used later by the sensor device to forward a reply message,
- *DADT action*, which specifies the name of the distributed action to be executed,
- *DADT Classname*, which provides name of the DADT instance,
- *DADT View* which represents the scope of the distributed request and used by the sensr device to select the relevant ADT instance.

The *Reply Message* used by sensor device to provide the requestor with relevant sensor data, if it is required. This message contains the following fields:

- *Source*, which specifies the sensor device sending the reply
- *Readings*, which stores sensor readings

These messages are sent over WSN and delivered to the nodes by means of LN, as it was discussed in Chapter 5.

6.4 The DADT/LN Prototype in the simulated environment

6.4.1 Simulation using JiST/SWANS

The DADT/LN prototype was tested on the SWANS WSN simulator, which is built upon the JiST discrete event simulator (see Section 4.1.2).

The DADT/LN prototype code is wrapped in the JiST API, and is loaded on to a simulated node. The DADT/LN prototype code did not require any changes before depolying onto simulated nodes due to design of the JiST/SWANS simulator.

As described in the previous section, there are two kinds of nodes in the DADT/LN prototype:

- *Controller Node*, which holds the distributed application, and the framework to manage the same.
- *Sensor Device*, which holds the individual ADT instances.

The controller node was implemented as a separate node on the JiST/SWANS simulator and was run as an independent simulated node. The sensor device implementation (see Section 6.2) was loaded on to all but one of the nodes in the simulator. Number of nodes in the simulated WSN varied from 30 to 100.

The controller node was programmed to perform requests for the distributed operation with a certain periodicity. Each sensor device declared 4 different ADT instances of type *Sensor*, e.g. Temperature sensor, Humidity senser, etc., and bound them to the DADT type *DSensor*.

The nature of discrete event simulation allowed to verify empirically the robustness of the prototype done as part of this thesis, but could not provide any evaluation results relevant for real-world WSNs.

6.4.2 The DADT/LN Prototype on Sun SPOTs

For further experimental validation of the implementation produced as part of this thesis, the DADT/LN prototype was deployed on Sun SPOTs [26] (see Section 4.2).

The Controller application was tested and executed to be run as either:

- an on-Host application which communicated with WSN the via basestation node, or
- an on-SPOT application that executed on one of the sensor devices in the WSN,

The other Sun SPOTs ran the Sensor Device implementation as on-SPOT applications (see Section 4.2.2 for a description of host and on-SPOT applications).

The deployment of the DADT/LN prototype on Sun SPOT devices involved several challenges. Whereas the JiST/SWANS simulator permits the execution of unmodified Java code on the simulated node, the Squawk JVM on Sun SPOTs introduces severe limitations on the code. The application code has to be compliant with the CLDC 1.1 specification of the Java ME framework, and therefore lacks support for a number of APIs that were previously used during the development of the DADT prototype [24].

Strict constraints on the supported classes and packages, e.g. *java.utils*, have required refactoring of the code when deployed on the Sun SPOT devices. Moreover, lack of support for the serialization of objects resulted in the necessity to perform serialization "manually" for the DADT Action and the DADT View objects contained in the request messages (see Section 6.3), and hence, increased size of the request message payload.

6.5 Summary

This chapter presented the details of the implementation of the DADT/LN prototype developed during the course of this work. The chapter concluded with a description of the challenges surmounted in order to further verify the DADT/LN prototype on the Sun SPOT sensor devices.

Chapter 7

Evaluation

This chapter presents the results of the performance evaluation of the DADT/LN prototype running on real-world sensor nodes (Sun SPOTs).

7.1 Setting

The sample distributed application used for evaluation of the DADT/LN prototype implements an example WSN application where several sensor devices are deployed in the environment and able to provide reports about some physical phenomena, e.g. temperature. A monitoring station may access particular sensor devices in WSN to aggregate their sensor readings, e.g. to calculate average value, or reset sensors in case of problems.

The sample distributed application initially consists of the following files:

- *Sensor.java*, which defines the functionality of the device that provides sensing ability.
- *SensorNode.dadt*, which defines the sensors and/or actuators available on the sensor device as abstract ADT instances, and binds them to the "DSensor" DADT type.
- *ClientNode.dadt*, which is run by monitoring station in order to perform the

distributed operation requests; the scope of the distributed operation is also declared here.

- *DSensor.dadt*, which declares the list of available distributed operations, actions and properties.

As described in Section **??** the dadt files are translated by DADT preprocessor (see Section **??**) into conventional java application. This results into creation of additional java files, each of which represents DADT Property and DADT Action classes (see Section 3.2.2) that were declared by application developer in *DSensor.dadt* file.

These files are named according the following naming scheme: "DSensorXXXproperty" or "DSensorYYYaction", where *XXX* is a name of the DADT Property and *YYY* is a name of the DADT Action respectively. Besides, if the description of DADT Properties is based on dynamically changed sensor properties, ADT instances of SensorNode map those into corresponding dynamic LN Predicates (see Section **??**), the additional files named as "LNProviderXXX" are to be added.

The functionality of the sample application is provided by the objects of the runtime library of the DADT/LN prototype **??**.

The code of the distributed application is coupled with the compiled runtime library to create 2 jar-files representing distributed applications, namely the SensorNode and ClientNode applications, which were deployed on the Sun SPOT devices.

6 Sun SPOT devices that form a simple WSN were used as a testbed for performing their evaluation. Each of the Sun SPOT devices ran one of the LN/DADT prototype distributed applications (ClientNode or SensorNode) application, as can be seen in Figure 7.1.

The sensor device that ran the SensorNode application used the temperature and light sensors available on the Sun SPOT demo sensor board. The DADT/LN prototype on the sensor device abstracted those sensors using ADT instances, and provided functionality to perform sensing and resetting operations. Communication between sensor devices was supported by the LN implementation.

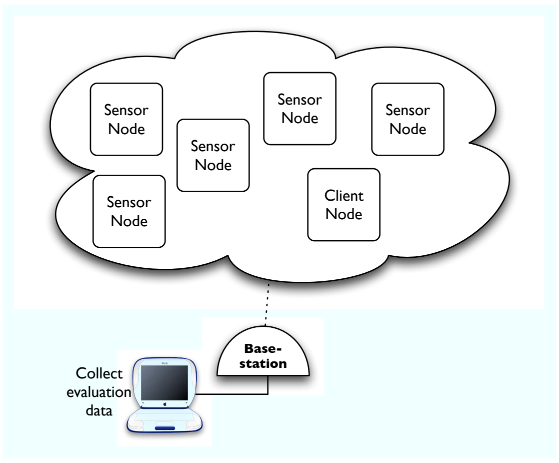

Figure 7.1: Test-bed model of WSN using the DADT/LN prototype

The sensor device that runs the ClientNode application was programmed to calculate the average temperature or luminosity value sensed by the set of nodes. This set of nodes was specified implicitly using a DADT Dataview object (see Section 3.2.2.2).

7.2 Metrics

The DADT/LN prototype was evaluated using the following metrics:

- Memory usage
- Processing and communication delays

These metrics were selected due to the fact that they allow to ..

7.3 Results

7.3.1 Memory Usage

As mentioned above, the memory requirements of the DADT/LN implementation are based on the following components:

Filename	DADT Source code, bytes	Java source code, Num of code lines Java byte-code size, bytes
Sensor.java	-	tbc 3164
SensorNode.dadt	477	tbc tbc
ClientNode.dadt	841	tbc tbc
DSensor.dadt	303	tbc tbc

Table 7.1: Memory usage

- Distributed application code translated and compiled from dadt files
- The runtime library of the DADT/LN prototype

Table 7.1 presents the memory requirements for the evaluated sample distributed application.

As discussed before, the runtime library consists of a number of Java files that provide the functionality of the DADT/LN prototype. The *jar* file representing the runtime library for the SUN SPOT device requires 81KB of memory.

The runtime library was later integrated with the SensorNode application code resulting in 95KB jar-file. Similarly, the integrated jar-file of the ClientNode application requires 99 KB of the memory.

7.3.2 Processing and Communication Delays

This section presents evaluation of the round trip delay time required for executing a distributed sensing operation on the WSN.

The processing and communication delays were used to compare the performance of the DADT/LN prototype upon execution of the distributed operation "average" over several such Dataviews.

The Dataview used to define the scope of the given distributed operation would be classified as follows:

- *low* complexity, which consists of up to 3 DADT properties allowing to define views, such as "all active temperature sensors";
- *moderate* complexity, which uses from 3 to 10 properties that allows to create views, such as "all active temperature sensors with precision ¿ 1.0 and sensor reading greater than 80";
- *high* complexity, which uses up to 20 properties. The number of properties that construct such type of Dataviews was limited by the length of the radiogram size used for communication between Sun SPOTs.

Figure 7.2 shows the relevant time delays occurred during the work of DADT/LN prototype.

7.3.2.1 Non-concurrent requests

This section provides results of the evaluation of the average time intervals on the basis of single non-concurrent requests.

As can be seen in Table 7.2, the complexity of the Dataview has a direct correlation to the delays associated with processing of the request on the SensorNode side. This subsequently leads to increasing waiting intervals on the ClientNode side.

On the SensorNode, this might be a consequence of the given limitations of the Squawk JVM, which does not provide support for serialization of objects.

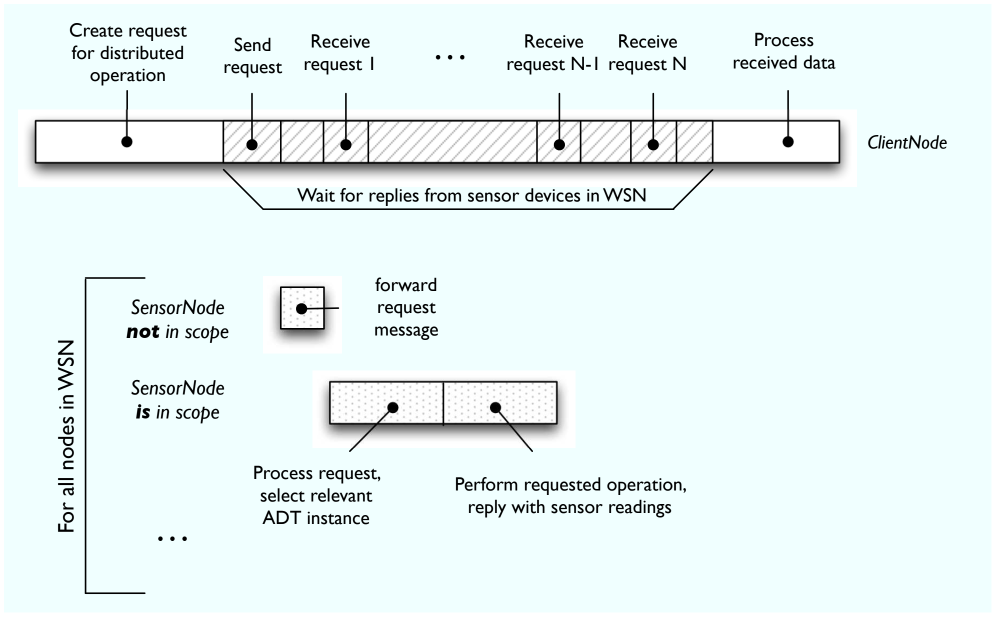
Create request for distributed operation
Send request
Receive request 1
. . .
Receive request N-1
Receive request N
Process received data
ClientNode
Wait for replies from sensor devices in WSN
For all nodes in WSN
SensorNode **not** in scope
forward request message
SensorNode **is** in scope
Process request, select relevant ADT instance
Perform requested operation, reply with sensor readings
. . .

Figure 7.2: Factors contributing to delays in the execution of a distributed operation on a WSN using the DADT/LN prototype

Hence, delays occur when the object representing the scope of the distributed operation is recreated by the SensorNode, and is later used for identifying the relevant ADT instances which execute the requested operation. Activities such as performing an action that involve: (a) requesting sensor readings, and (b) the subsequent invocation of the LN API to send a reply message, were executed in a relatively short time.

On the ClientNode, actions such as preparing a request message and parsing the received results take on average 10-15% of the time spent waiting for the receipt of reply messages with sensor readings.

7.3.2.2 Concurrent requests

Table 7.3 shows the processing time delays caused by concurrent requests on the WSN. The WSN used as part of this evaluation consisted of two devices running the Client Node application, which issued concurrent distributed operation execution requests. Evaluation of the concurrent requests was based on the assumption

	ClientNode Application			SensorNode Application	
Dataview complexity	Create request (ms)	Send request over LN and wait for replies (ms)	Process sensor readings (ms)	Receive request, select ADT instances	Perform requested action and send reply
Low	44.64	1414.27	203.91	520.13	92.94
Moderate	49.82	3101.91	203.45	943.02	93.76
High	62.55	6034.82	205.45	1077.83	89.95

Table 7.2: Processing and communication delays for non-concurrent requests for execution of distributed operation “average”

that these requests may be 10-80 ms apart in time.

ClientNode Application			SensorNode Application		
Dataview complexity	Create request, ms	Send request over LN and wait for replies, ms	Process sensor readings, ms	Receive request, select ADT instances	Perform requested action and send reply
Low	51.40	2604.75	197.85	595.95	178.84

Table 7.3: Processing and communication delays for concurrent requests for execution of distributed operation “average”

The results presented in Table 7.3 considered only those Dataviews with low complexity. For concurrent requests over Dataviews with a high-level of complexity, the time delays were found to be inappropriately large for executing a distributed operation.

It can be seen that, in general, concurrent requests increase the processing time on devices running the SensorNode application. However, the other delay factors involved in the execution of the distributed operation are comparable to that incurred upon execution of non-concurrent requests.

Future work may include further evaluation and adaptation of the DADT/LN prototype to provide acceptable performance for Dataviews of moderate and high complexity containing of a large number of DADT properties.

Chapter 8

Conclusions, and Future Work

8.1 Conclusions

During the course of this project, it was proven that the concept of DADTs can be applied to real world WSNs. The proof-of-concept prototype implemented indicates the potential of programming abstractions in helping reduce the effort involved in developing applications for WSNs. In addition, upon integrating the innovative LN routing mechanism with the DADT implementation on the application layer, there was found to be a significant reduction in the amount of processing performed in the application layer.

8.2 Future Work

This section presents a list of possible extensions to the work implemented as part of this thesis. These include:

- *Support for DADT selection operators:* The current prototype supports the selection of all ADT instances that match a defined DADT Data view, but does not enable the selection of a subset of the aforementioned collection of ADT instances. This arises from the limitations of the current LN implementation.

- *Extending support for Space DADTs:* Currently, the prototype provides a limited support for the notion of space. Therefore, a possible avenue for future work could include the full support for Space DADTs provided by the prototype.
- *Extending the prototype for networks of heterogenous nodes:* The current prototype, by virtue of it being implemented in Java, cannot be used on a wide variety of different nodes.

Bibliography

[1] National institute of standards and technology. http://www.nist.gov.

[2] Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., et al. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. *IEEE ICDCS* (2004).

[3] Akyildiz, I., W.Su, Y. Sankarasubramanian, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine* (August 2002), 102–114.

[4] Andrew S. Tannenbaum. *Computer Networks*. Prentice Hall PTR, 2002.

[5] Bakshi, A., Prasanna, V., Reich, J., and Larner, D. The Abstract Task Graph: a methodology for architecture-independent programming of networked sensor systems. *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services* (2005), 19–24.

[6] Barr, R. SWANS-Scalable Wireless Ad hoc Network Simulator Users Guide, 2004.

[7] Barr, R., Haas, Z. J., and van Renesse, R. JiST: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper. 35*, 6 (2005).

[8] BONNET, P., GEHRKE, J., AND SESHADRI, P. Towards sensor database systems. *Proceedings of the Second International Conference on Mobile Data Management 43* (2001).

[9] BUONADONNA, P., HILL, J., AND CULLER, D. Active message communication for tiny networked sensors. *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM01)* (2001).

[10] FOK, C.-L., ROMAN, G.-C., AND LU, C. Mobile agent middleware for sensor networks: an application case study. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (Piscataway, NJ, USA, 2005), IEEE Press, p. 51.

[11] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Notices 38*, 5 (2003), 1–11.

[12] GUMMADI, R., GNAWALI, O., AND GOVINDAN, R. Macro-programming wireless sensor networks using Kairos. *Intl. Conf. Distributed Computing in Sensor Systems (DCOSS)* (2005).

[13] GUTTAG, J. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (1977), 396–404.

[14] HADIM, S., AND MOHAMED, N. Middleware: middleware challenges and approaches for wireless sensor networks. *Distributed Systems Online, IEEE 7*, 3 (2006).

[15] HAN, C., RENGASWAMY, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. SOS: A dynamic operating system for sensor networks. *Third International Conference on Mobile Systems, Applications, And Services (Mobisys)* (2005).

[16] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev. 36*, 5 (2002), 85–95.

[17] LUO, L., ABDELZAHER, T., HE, T., AND STANKOVIC, J. EnviroSuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS) 5*, 3 (2006), 543–576.

[18] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS) 30*, 1 (2005).

[19] MOTTOLA, L., AND PICCO, G. Logical neighborhoods: A programming abstraction for wireless sensor networks. *Proc. of the the - Springer 2* (2006).

[20] MOTTOLA, L., AND PICCO, G. Programming wireless sensor networks with logical neighborhoods. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks* (New York, NY, USA, 2006), ACM.

[21] MOTTOLA, L., AND PICCO, G. Using logical neighborhoods to enable scoping in wireless sensor networks. In *MDS '06: Proceedings of the 3rd international Middleware doctoral symposium* (New York, NY, USA, 2006), ACM.

[22] MOTTOLA, L., AND PICCO, G. P. Programming wireless sensor networks: Fundamental concepts and state-of-the-art. University of Trento, Italy.

[23] NEWTON, R., MORRISETT, G., AND WELSH, M. The regiment macroprogramming system. *Proceedings of the 6th international conference on Information processing in sensor networks* (2007), 489–498.

[24] PICCO, G., MIGLIAVACCA, M., MURPHY, A., AND G., R. Distributed abstract data types. In *Proceedings of the 8th International Symposium on*

Distributed Objects and Applications (DOA'06) (2006), R. Meersman and Z. Tari, Eds., vol. 4276 of *Lecture Notes in Computer Science*, Springer.

[25] SHANKAR, A. U. Discrete-event simulation. Tech. rep., Department of Computer Science, University of Maryland, January 1991.

[26] SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. *Proceedings of the 2nd international conference on Virtual execution environments* (2006), 78–88.

[27] SUN MICROSYSTEMS, INC. *Sun(TM) Small Programmable Object Technology (Sun SPOT) Developer's Guide*, July 2008.

[28] WELSH, M., AND MAINLAND, G. Programming sensor networks using abstract regions. *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation-Volume 1 table of contents* (2004), 3–3.

[29] YAO, Y., AND GEHRKE, J. Query processing for sensor networks. *Proceedings of the 2003 CIDR Conference* (2003). Department of Computer Science Cornell University.

[30] YU, Y., KRISHNAMACHARI, B., AND PRASANNA, V. Issues in designing middleware for wireless sensor networks. *IEEE Network* (2004), 16.