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

Enabling Distributed Programming Abstractions for Real-World Wireless Sensor Networks (draft)

Relatori

Gian Pietro Picco
Klaus Wehrle

Laureanda

Galiia Khasanova

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

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 [**?**].

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 operation of WSNs. This limits its utility in widely disparate fields by scientists unaware of the technology underlying them. To deal with this problem, it is necessary to abstract the details of WSN operation thereby allowing the application programmer to develop WSN applications inspite of being naive to the details of embedded system programming. 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) [**?**] 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 (or a subset thereof).

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 [**?**] to WSNs.
- It integrates the DADT mechanism in the application layer with a Logical Neighbourhood [**?**] 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 on simulations as well as real nodes.

1.3 Outline

The outline of the thesis is as follows: Chapter 2 presents an introduction to wireless sensor networks, and discusses the protocol stack used in a typical WSN application and a short survey of routing mechanisms for WSNs. Chapter 3 discusses the background underlying this work, and includes brief descriptions of WSN programming models, DADTs, the LN routing mechanism, and the hardware platform used to evaluate the prototype produced as part of this work. Chapter 4 describes the details of the implementation of the prototype as well as a brief description of the structure of the DADT prototype that was modified during the course of this work. Finally, Chapter 5 describes the metrics and experimental methodology used to evaluate this work, the conclusions thus reached, and possible avenues for future work.

Chapter 2

Introduction to Wireless Sensor Networks

This chapter presents a brief introduction to Wireless Sensor Networks (WSNs). Additionaly, 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.

A sensor node consists of the following elements (as can be seen in Figure 2.1)

- *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.

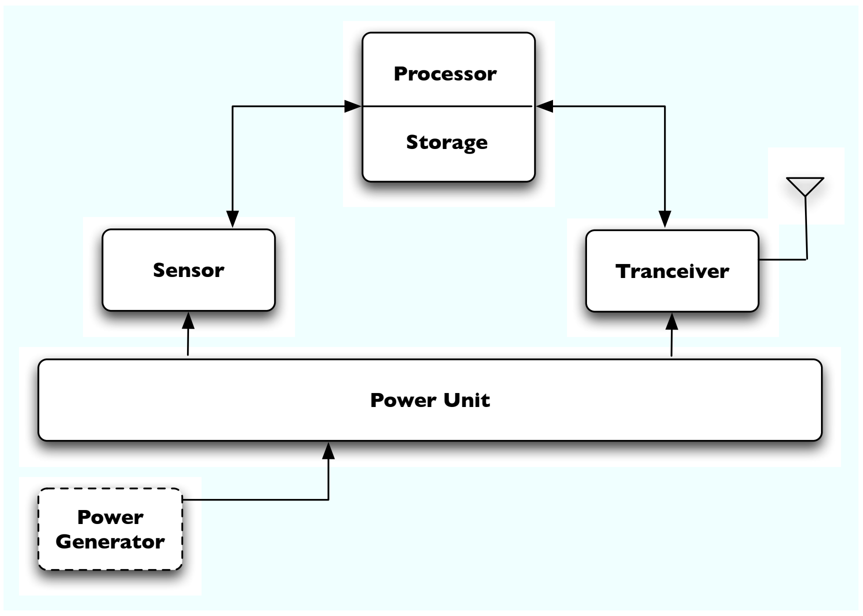

Figure 2.1: Architecture of a sensor node (adapted from [**?**]).

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 [**?**]:

- *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, 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 [**?**] is an adaptation of a generic protocol stack [**?**]. As can be seen from Figure 2.2, 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.

2.3 Routing in WSNs

According to [**?**], routing algorithms may be classified on the basis of the network structure used. Using this metric, routing algorithms are classified along the or-

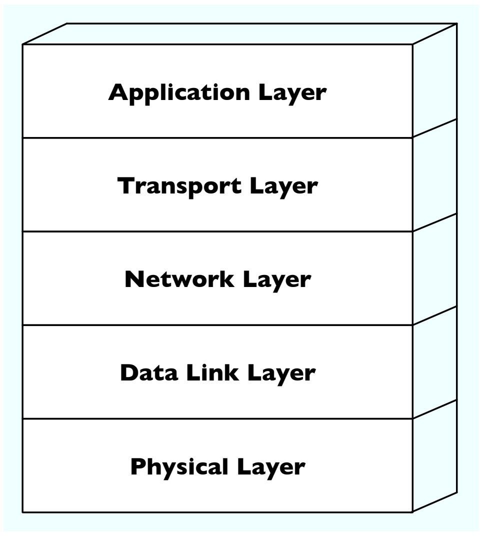

Figure 2.2: WSN protocol stack (reproduced from [**?**])

ganisation used for the WSN. Routing algorithms are thus primarily classified into being either:

- *Flat*, where every node plays the same role,
- *Hierarchical*, where the network is organised into physically defined clusters,
- *Location-based*, where positioning information is used to direct traffic to a given physical subset of the network.

Additionally, depending on how the source finds a route to the destination routing protocols can be split into three categories [**?**]:

- *Proactive*, when all routes are known before they are used
- *Reactive*, when route is computed on demand
- *Hybrid*, which uses a combination of the two techniques above.

The LN routing algorithm that is principal to this work may be considered an extension of a location-based, hybrid routing algorithm, as logical predicates are used to direct traffic to a specific *logical* subset of the network. A more detailed discussion of the LN mechanism may be found in Chapter 3.

Chapter 3

Background

this chapter should be checked

This chapter presents a brief discussion of the concepts, algorithms, and hardware platforms that were used during the course of this work. This chapter begins with an introduction to programming abstractions, and continues with a discussion of their applications in WSN programming.

This chapter further presents the concepts underlying Distributed Abstract Data Types (DADTs). This is followed by a presentation of the Logical Neighborhoods (LN) [**?**], a mechanism that enables routing and scoping in WSNs. The chapter then concludes with a description of the hardware platform - Sun Small Programable Object Technology (SPOT) [**?**] - used during the course of this work to experimentally validate the implemented prototype in a real-world environment.

3.1 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 a strict limit of resources on the sensor node, as it was discussed in the 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 be used [**?**].

To increase the ubiquity of WSN usage, it is essential that the 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 which engineer a shift in focus towards the system and its results, as opposed to sensor node functionality itself [**?**].

According to Yu et al [**?**], 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.

3.1.1 Taxonomy of WSN Programming Models

Existing programming models for WSNs cover different areas and can serve different purposes. They can be classified into two main types, depending on the applications they are used for [**?**] (see Figure 3.1):

- *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 [**?**], Cougar [**?**], SOS [**?**], Agilla [**?**].

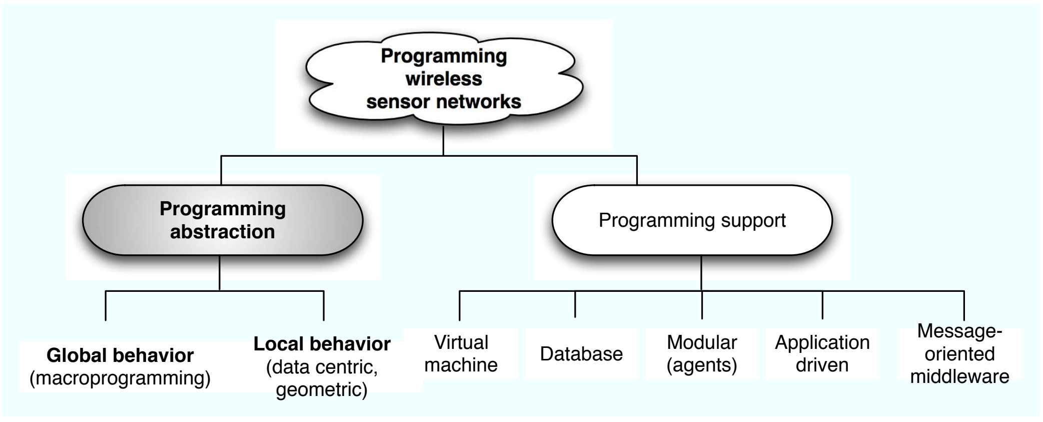
Programming wireless sensor networks
Programming abstraction
Programming support
Global behavior (macroprogramming)
Local behavior (data centric, geometric)
Virtual machine
Database
Modular (agents)
Application driven
Message-oriented middleware

Figure 3.1: Taxonomy of WSN programming models (reproduced from [**?**])

- *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 TinyOS [**?**], Kairos [**?**] and EnviroTrack [**?**].

The rest of this section focuses on the discussion of WSN programming abstractions, as it is this type of programming model that is relevant to the work presented herein.

3.1.2 Classification of WSN Programming Abstractions

Programming abstractions may either be *global* (also referred to as macroprogramming) or *local* [**?**].

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

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.

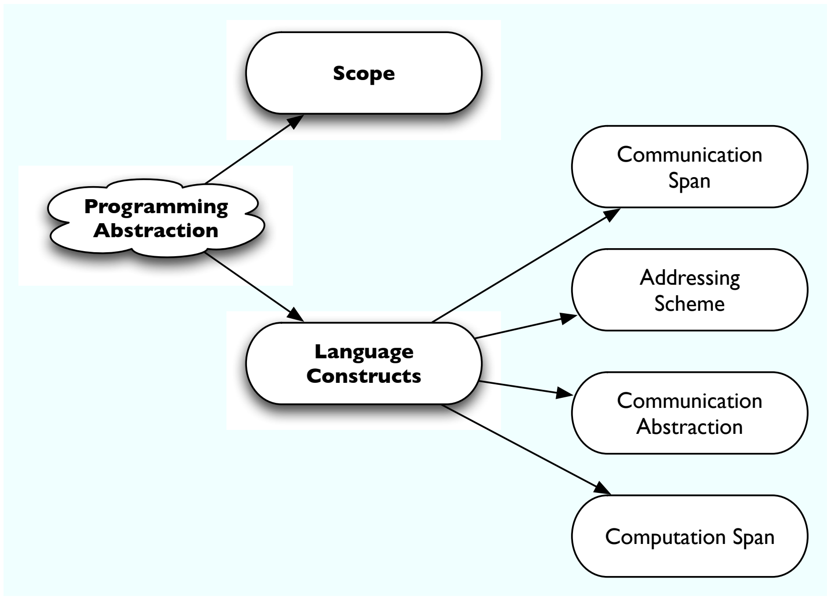

Figure 3.2: Classification of Programming Abstractions

Programming abstractions may also be classified on the basis of the nature of the language constructs made available to the WSN programmer [**?**]. Some of the metrics used for classification are [**?**]:

- *Communication span*
- *Addressing scheme*
- *Communication abstraction*
- *Computation span*

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

3.1.2.1 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:

- *Physical neighbourhood*
- *Multi-hop group*
- *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.

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 such path exists.

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.

3.1.2.2 Addressing scheme

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

- *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.

When *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 nodes at different times.

3.1.2.3 Communication Abstraction

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

- *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 performs communication using high-level constructs.

3.1.2.4 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:

- *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.

As an illustration of the communication span of type *Group*, or even possibly *Global*, could be an example of the WSN programming abstraction that allows to send a message to all nodes in WSN, requiring to perform reset of the node, if its sensor readings exceed a specific theshold.

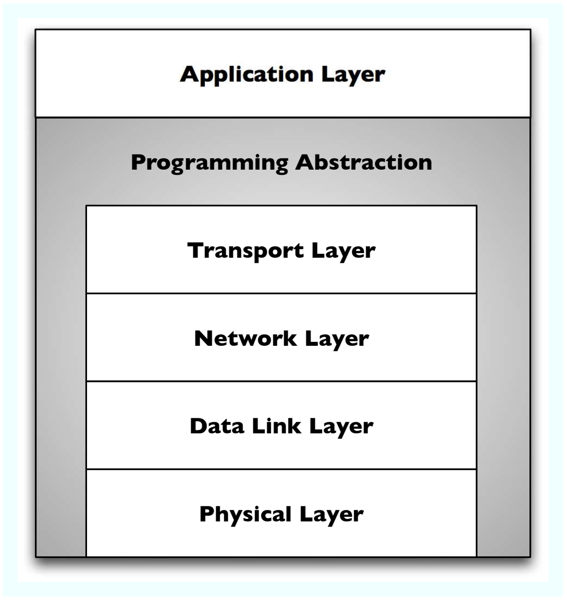

Figure 3.3: Programming models on the WSN protocol stack (adapted from [?])

3.1.3 Programming models on the WSN protocol stack

WSN programming models 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 3.3, 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.

3.2 Distributed Abstract Data Types

Distributed Abstract Data Types is a new programming language construct used to support distributed and context-aware applications. The concept of DADTs was introduced in [**?**]. The rest of this section discusses the concepts and provides the reader with the relevant background and examples[1].

3.2.1 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 usually 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 [**?**]. From the the application developer's point of view, use of ADTs allows to separate interfaces from the specific implementations.

One can consider *stack* as a simple example of an ADT [**?**]. 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 from the proposed specification.

The concept of ADTs has been sucessfully used in the different areas of science. The rest of this section focuses on the application and extension of the concept of ADTs for use in WSNs.

3.2.2 ADTs in WSNs

A WSN as described earlier, usually consists of a number of sensor node. Each sensor node may include several sensors.

ADT instance *Sensor* can be used to abstract different types of sensor that might be available on the 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 presented by *Sensor Node* in the Figure 3.4, and be later used by the application developer[2].

[1]Provided examples are adapted from [**?**]

[2]Further details of ADT specification and intantiation is provided in the Section 4.1.1

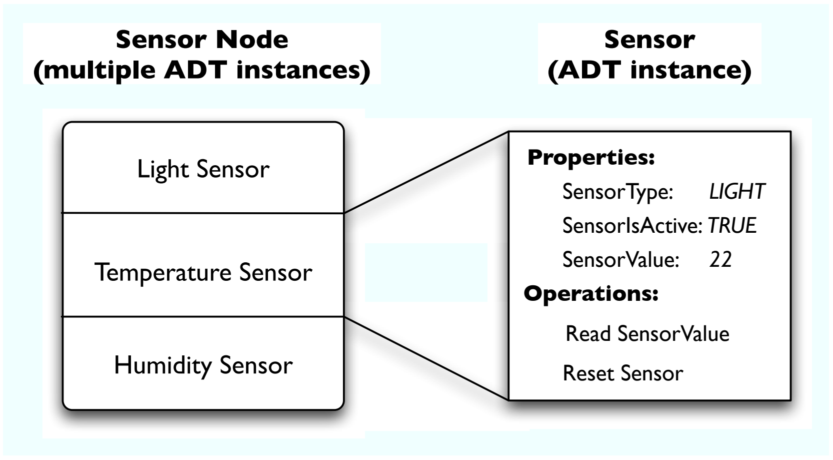

Figure 3.4: Abstraction of sensor node through multiple ADTs

3.2.3 Data and space ADTs

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

3.2.4 DADTs as an extension of ADTs

An extension of ADTs - a class of Distributed ADTs (DADTs), have applications in distributed programming models. The state of multiple homogenious ADTs in a distributed system are made collectively available using the interface of a DADT [**?**].

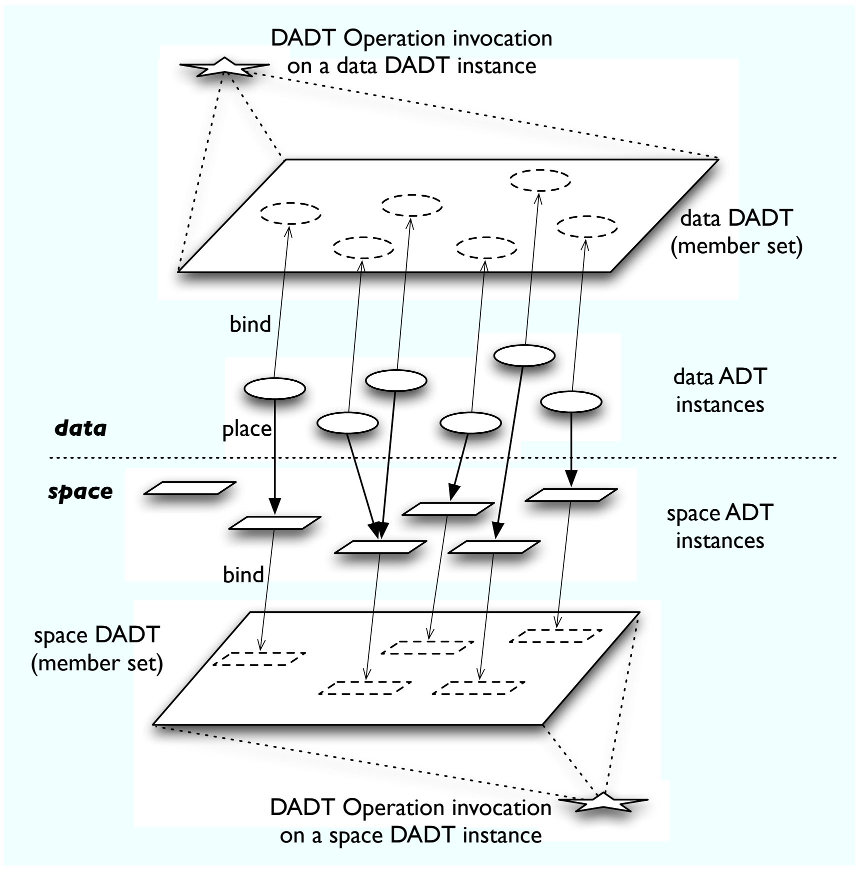

Figure 3.5: Data and space in the DADT model (reproduced from [**?**])

DADTs can be used when the distributed state of the system is the primary issue of importance. 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 3.5):

- *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.

3.2.4.1 DADT Operators and Actions

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

DADT operators may belong to one of the following types:

- *Selection Operators* that allow to perform distributed operation on a subset of the instances in the member set.
- *Conditional Operators* that provide support for global conditions to be applied on the member set before execution of the DADT operation.
- *Iteration Operators* that allow iterating over the ADT instances in the set, and thus permit access to individual ADT instances.

DADT actions are special DADT programming constructs. They are declared in the DADT type, but are executed as an operation on remote ADT instances.

3.2.4.2 Views

DADT Views allow to defines 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 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 executed locally on the ADT instance [**?**]. DADT View may either be:

- *Data View*,

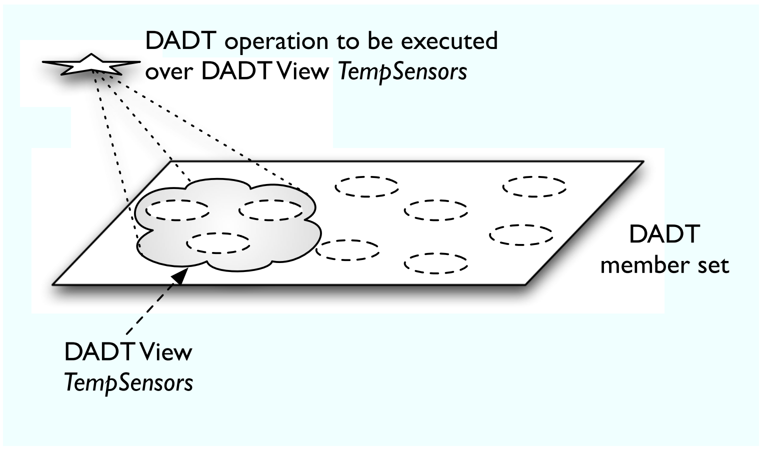

Figure 3.6: DADT views (reproduced from [**?**])

- *Space View*.

The concept of DADT Views is presented in Figure 3.6.

3.3 Logical neighbourhoods

Typically, communication between WSN nodes is based on routing information between nodes by exploiting the communication radius of each node. Thus, 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 this.

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 example. 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 a recent history of recorded motion. The sensors at the entrance form a logical neighbourhood in this case. However, as the entrances may be widely separated, it is not necessary that this subset of nodes is part of a single physical neighbourhood. The use of current WSN programming techniques to enable a mechanism of this nature entails additional programming effort, be-

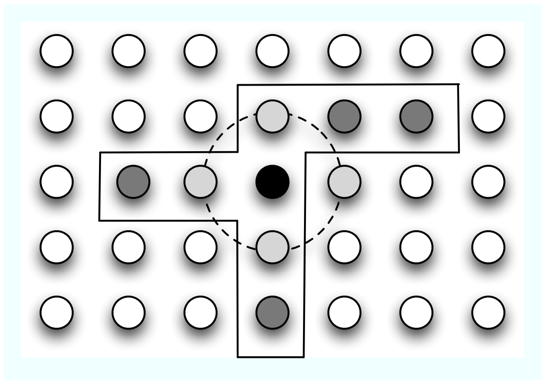

Figure 3.7: Representation of logical and physical neighborhood of a given node. *The dashed circle represents the physical neighborhood, whereas the solid polygon represents (one of) the node's logical neighborhood* (reproduced from [**?**])

cause the developer has to deal not only with the application logic, but also with identifying the system portions to be involved and ways those should be reached, which might lead to an increased complexity of the code [**?**].

Mottola and Picco [**?**] suggest the addressing of the mentioned above issues by using *Logical Neighbourhoods (LNs)*, an abstraction that replaces the node's physical neighbourhood with a logical notion of proximity (See and Figure 3.7). Using this abstraction, programmers can communicate with members of a LN using a simple message passing API, thereby allowing for logical broadcasts. The implementation of this API is supported by means of a novel routing mechanism devised specifically to support LN communication.

The rest of this section discusses the LN abstractions and provides further details of the underlying routing mechanism.

3.3.1 The LN Abstraction

LNs can be specified using a declarative language such as SPIDEY [**?**, **?**][3], and involves the definition and instantiation of the *node* and the *neighbourhood*.

[3]The LN implementation used as part of this work does not use a declarative language for LN specification

Nodes are a logical representation of the subset of a sensor node's state and characterstics that are used for the specification of an LN. Nodes are defined in a node template and are subsequently instantiated, as shown in Listing 3.1

```
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 3.1: Node Definition and Instantiation

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.

Listing 3.2 shows the definition and instantiation of a neighbourhood - based on the node template defined in Listing 3.1 - 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 3.2: Neighbourhood Definition and Instantiation

Communication using LNs is performed using a simple API which overrides the traditionally used broadcast facility and makes it dependent on the (logical) neighbourhood the message is addressed to [**?**]. The routing mechanism described in the next section enables LN commmunication.

3.3.2 LN Routing

The routing approach used in LNs is structure-less, and uses a local search mechanism based on a distributed state space built by the periodic update of node profiles. The routing mechanism can be divided into two phases [**?**]:

3.3.3 Construction of a distributed state space

Each node periodically transmits a *profile advertisement* that contains information on the attribute-value pairs defined in the node template. This message causes an update in its physical neighbours' *State Space Descriptors (SSDs)* if (a) no entry exists for any given attribute-value pair specified in the profile advertisement, or (b) the *transmit cost* is lower than the costs for any existing SSD entry for a particular attribute-value pair. If any such change occurs, the profile advertisement is rebroadcast with an updated cost.

Additionally, passive listening to profile advertisements for attribute-value pairs with higher costs than what is entered in the SDD is used to construct increasing paths.

3.3.4 Routing through Local Search

When a message has to be sent to a particular LN, the sending node transmits the message to any node in its SSD whose attribute-value pairs match the LN predicates. The message is associated with a specific set of *credits* from which the cost to send to the node is deducted. This process continues until the message is received at node(s) in the LN along the reverse of the path, called a *decreasing path*, which is determined during the first phase.

However, to ensure that messages are received by every nodes that belongs to the neighbourhood (and the algorithm is not trapped in local state-space minima), *exploring paths* are used at specific points during the message traversal (at nodes which meet the neighbourhood predicates, and/or after a user-defined number of hops). The credit reservation system described above is used in that case, with the node dividing the reserved credits between following decreasing paths and exploring paths.

3.4 Sun SPOTs

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

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.

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

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

However, Simon et al [**?**] 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, as well as developer productivity.
- Opportunity to use standard development and debugging tools.

Figure 3.8: Sun SPOT device

3.4.1 The Sun SPOT hardware platform

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 3.8) 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.

3.4.2 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 [**?**]:

- It is written in Java, and specifically designed for resource constrained devices, meeting the Connected Limited Device Configuration (CLDC) Java Micro Edition (Java ME) configuration requirements.

- It does not require an underlying OS, as it runs directly on the Sun SPOT hardware, which allows to reduce memory consumption.
- It suuports inter-device application migration.
- It allows the execution of multiple applications on one VM, representing each one as an object.

3.4.3 Split VM Architecture

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 it is shown in the Figure 3.9.

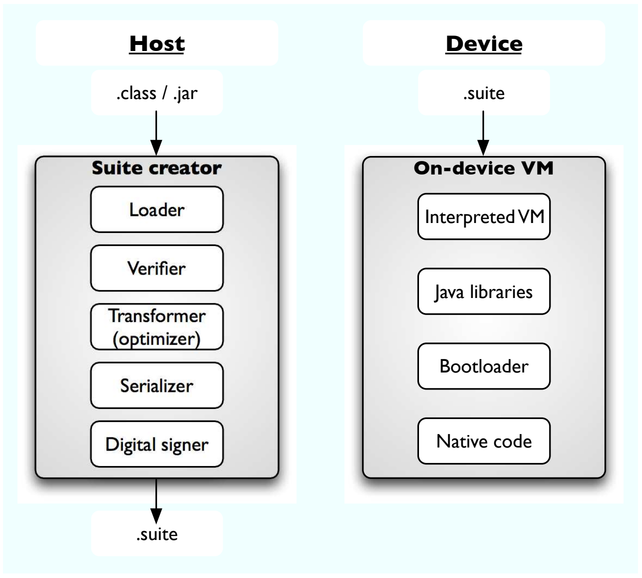

Figure 3.9: The Squawk Split VM Architecture (reproduced from [**?**])

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 [**?**], Squawk bytecodes are optimised in order to:

- minimise space used by using smaller bytecode representation, escape mechanisms for float and double instructions, and widened operands.
- allow for 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 storing on the operand stack only the operands for 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.

3.4.4 Sun SPOT applications

Sun SPOT applications are divided into two classes: [**?**]:

- *On-SPOT applications*
- *On-Host applications*

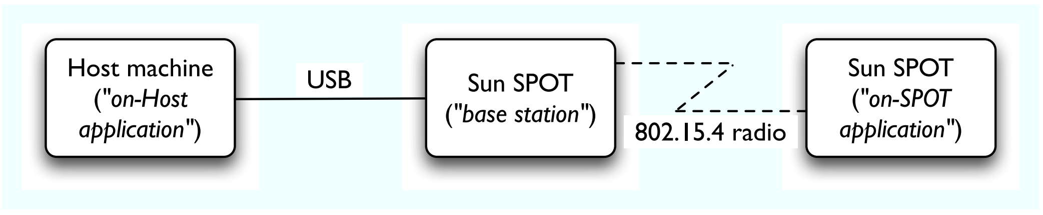
Host machine ("*on-Host application*")
USB
Sun SPOT ("*base station*")
802.15.4 radio
Sun SPOT ("*on-SPOT application*")

Figure 3.10: Types of Sun SPOT applications (adapted from [**?**])

On-SPOT applications are are deployed and executed on a remote Sun SPOT that communicates untethered. On-SPOT applications is a Java program that runs on the Squawk VM, and is compliant with Java ME.

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 3.10). The host application is a Java 2 Standard Edition (J2SE) program.

Some conclusions about what has been discussed learned and what is next

The LN mechanism that forms the bedrock of this work (see Section 3.3) provides the functionality of the Data Link Layer and the Network Layer. The concept of Distributed Abstract Data Types (see Section 3.2) is restricted to the application layer, and the focus of this work thus lies primarily within the application layer.

Chapter 4

Distributed Programming Abstractions for WSNs

This chapter presents the details of the DADT/LN prototype implemented as part of this work. The chapter begins with a description of the DADT prototype [**?**], and outlines its limitations. This is followed by a discussion on the architecture of the hybrid DADT/LN prototype th combines the DADT prototype with the LN routing mechanism, and extends the existing prototype to work on WSNs. This is followed by a description of the JiST/SWANS simulation environment used to verify the implementation, and the experimental validation performed using the SunSPOT hardware platform.

4.1 The DADT Prototype

The DADT prototype implemented by Migliavacca et al [**?**] 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[1], a description of the existing DADT prototype [**?**], and a discussion on its limitations.

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

4.1.1 ADTs Specification and Instantiation

As was mentioned in Section 3.2.2, each sensor node in a WSN can be abstracted using multiple ADT instances (see Figure 3.4), and therefore conceal the details of sensor node abstraction from the application developer. The code snippets below are provided to further illustrate these concepts.

The specification for a described ADT[2] may be defined as a Java interface, as shown in Listing 4.1

```
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 4.1: Sensor ADT instances

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

- 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

and operations that allow to *Read* sensor data and to *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 -

[2]This is an example of the Data ADT (see Section 3.2.3)

(a) a temperature sensor and (b) a light sensor - may be defined using the ADT specification described as shown in Listing 4.2.

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

Listing 4.2: Sensor ADT instances

4.1.2 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 instantiaton.

4.1.2.1 Specification

DADT specifications can be best understood by carrying forward the example described in Section 4.1.1. To allow for collective access to multiple ADT instances of the type specified in Listing 4.1, a DADT *DSensor* may be defined as shown in Listing 4.3.

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

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

Listing 4.3: Data DADT specification (reproduced from [**?**])

The 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 3.2.4.2).

The *average()* operation allows to calculate the average of readings of every sensor in the member set or subset defined by DADT view.

Additionally, the DADT specification shown in listing 4.3 declares two DADT Properties, which will be discussed shortly.

4.1.2.2 Instantiation

Listing 4.4 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 4.4: DADT Instantiation (reproduced from [**?**])

4.1.3 Binding ADTs to DADTs

As mentioned earlier in Section 3.2.4, a DADT member set consists 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 4.5, where the Sensor ADT (see Listing 4.1) is bound to the DADT type *DSensor* defined in Listing 4.3.

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

Listing 4.5: Binding ADT instances to a DADT instance

4.1.4 Implementing DADT Operators and Actions

The concepts of DADT Operators and Actions were introduced in the previous chapte (see Section 3.2.4.1). This section describes the implementation of Operators and Actions in the DADT prototype [**?**].

The use of the DADT selection operator *all* can be understood by considering an extension to the example first introduced in Listing **??**) (see Listing 4.6).

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

Listing 4.6: Use of DADT Selection Operator

The distributed operation *resetAll* uses the selection operator *all* in order to obtain 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 (see Listing 4.1).

However, the application developer may in some situations be constrained by the available operations defined in the ADT specification. One of the solutions could be use of DADT actions, which is defined in the DADT type, but executed entirely on the ADT instance. THIS DOES NOT MAKE SENSE. EXPLAIN IT TO ME - SID

The mplementation of the distributed operation *average()* may require to use not a simple operation *read* provided by an ADT instance, but more complicated *reliableRead* action that capable of peforming extra measures. For instance, if ADT instance can notify about read failure by returning *ERROR* value, it could be convenient to prevent any transient faults of the sensor by performing a number of subsequent readings, and in case of failure simply reset the sensor, and read again. This behaviour can be defined as an action in *DSensor* class as shown in Listing 4.7. THIS PARAGRAPH IS UNNECESSARY DETAIL - WE DONT

NEED IT!!!!

It is important to note that, though DADT actions are defined in the DADT type, they are executed *locally* on the ADT instances.

```
THIS IS UNNECESSARY DETAIL!!!
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();
    ...
    // execution of the average value based on received readings
    ...
    }
}
```

Listing 4.7: Use of DADT Action (reproduced from [**?**])

4.1.5 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 3.2.4.2, DADT views are created by 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 4.8, can be delared.

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

Listing 4.8: Definition of DADT Data View

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

Listing 4.9: Definition of DADT Properties

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 contains only sensor nodes with temperature sensors for which the evaluation of both DADT properties (See Listing 4.9) *isActive* and *isSensorType* return *true*.

4.1.6 Limitations of the DADT Prototype

The DADT prototype proposed in [**?**] 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 on the JVM[3]. It provides support for DADT constructions and methods, such as *binding* ADTs to DADTs, *DADT Views*, *Actions* and *Operators*.

The communication in the prototype is based on the IP Multicast principle, and allows to deliver information to ADTs bound to a specific DADT, defined as a multicast group.

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, and besides has the following major limitations:

- The lack of a routing mechanism.
- Limitations in portability to real WSN nodes.
- The use of IP multicast for communication is not efficient on WSNs that have simpler protocol stacks. CHECK THIS!

4.2 The DADT/LN Prototype

4.2.1 Motivation

The DADT prototype proposed in [**?**] proved that the concept of DADTs could possibly be applied to WSN applications, but existing limitations with 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.

[3]The prototype runs on the full JVM, and not the Squawk JVM, which the DADT/LN prototype is capable of running on.

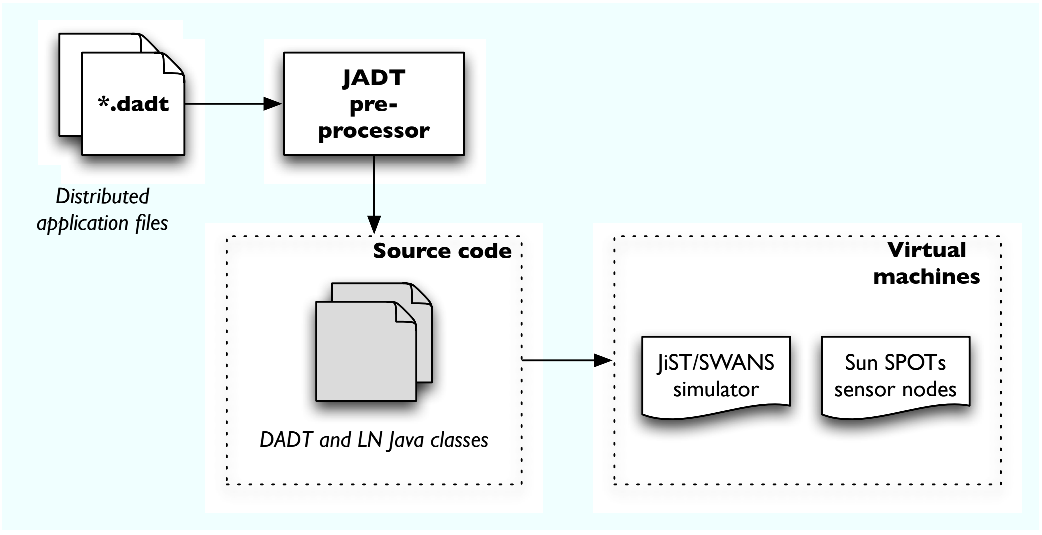

Figure 4.1: Workflow for development of an application that uses the DADT/LN prototype

- Interfacing the LN mechanism presented in [**?**] 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.

4.2.2 Overview

The overview of the workflow involved in using DADTs to enable WSN application programming is shown in Figure 4.1. The application developer provides an application layer code for the WSN using a DADT language in a series of *.dadt* files, that is Java code extended by DADT constructs. The JADT preprocessor is 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 [**?**]). In order to facilitate routing to LNs defined by DADT Views, the DADT infrastructure is interfaced with the a previously developed implementation of LNs.

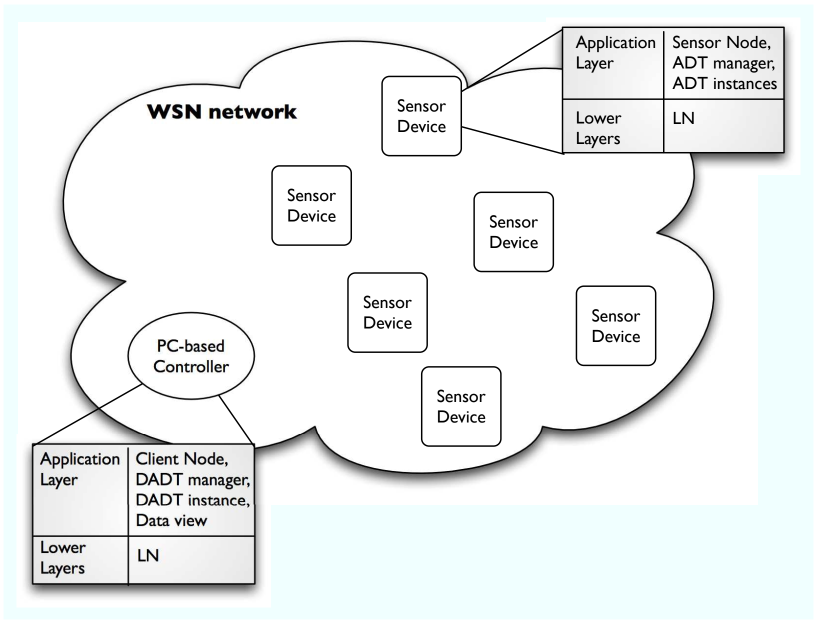

Figure 4.2: Schematic representation of the WSN as abstracted in the DADT/LN prototype

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

- the JiST/SWANS simulator [**?**, **?**] (See Section 4.2.5.1 where implementation details of the simulator are provided)
- a collection of Sun SPOT wireless sensor devices [**?**] (see Section 3.4) 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 4.2:

- *Controller*
- *Sensor Device*

Controller typically resides on a PC. 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.

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

4.2.3 Architecture

The architecture of the DADT/LN prototype is presented in Figure 4.3, and consists of the following logical layers.

4.2.3.1 Upper layer

The upper layer consists of the following components:

- *Distributed application*
- *Sensor* and *DSensor*
- *DADT runtime layer*
- *DADT manager*

The Distributed application comprises the code implemented by the application developer, and is written in Java, extended with constructs defined in the DADT specification language. This code is later translated into executable Java code using a preprocesssor and interfaced with the DADT runtime.

— DONE TILL HERE

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 ADT *Host* and space DADT *Network* are (currently - CHECK THIS) built into the DADT/LN prototype.

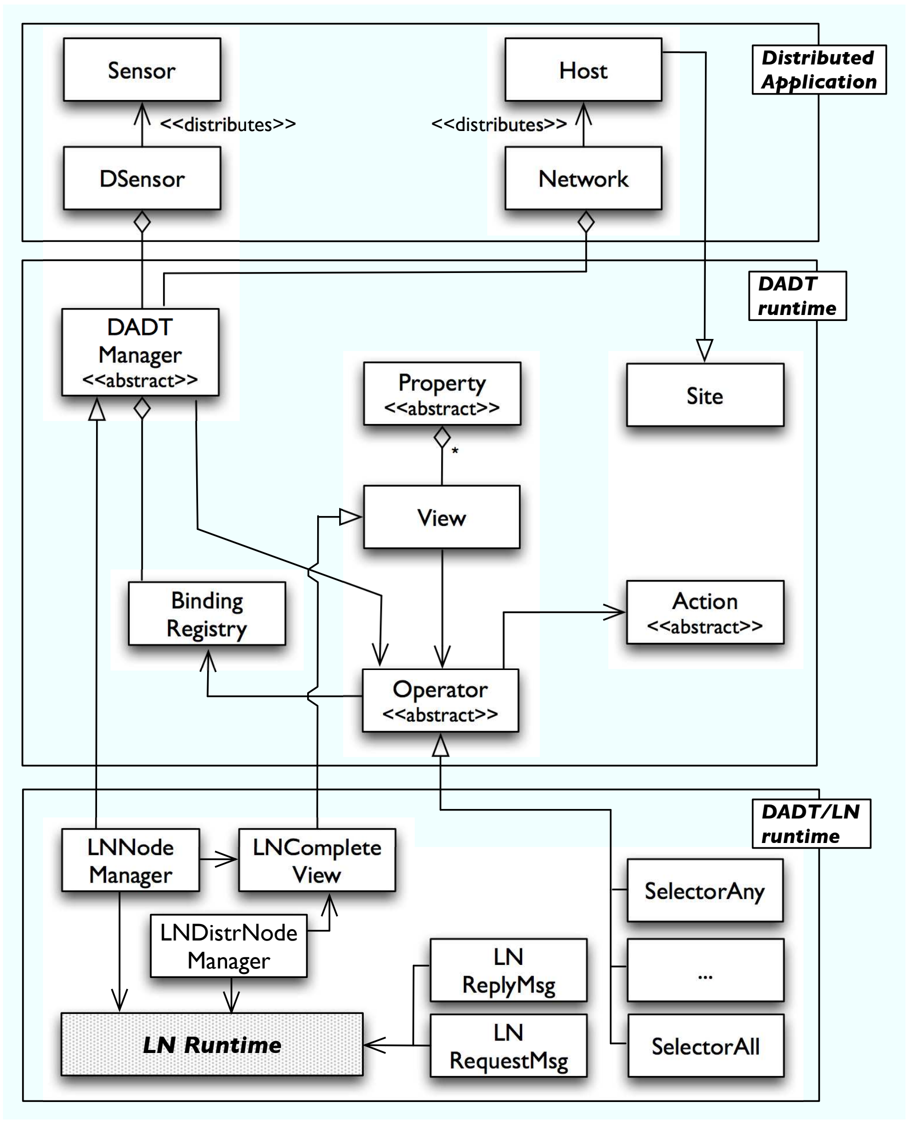

Figure 4.3: Architecture of the DADT/LN prototype

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

The *DADT Manager* manages the binding and unbinding of ADT instances to DADT type by means of the *Binding Registry* class, provides support for space ADTs using the *Site* class, and handles other relevant aspects WHAT RELEVANT ASPECTS?!!!.

— DONE TILL HERE

The abstract classes *Property* and *Action* represent the corresponding concepts, described in the Section 3.2.4. The class *View* provides support for data and space DADT Views, and contains the set of *Property* objects, that define the scope of the operations. Properties are organised into an abstract tree which provides the logical predicate defining the view, with the leaves representing DADT properties and the nodes specifying boolean operators used to compose it. The class *Operator* is a superclass for any of the DADT Operators, though current implementation of the DADT/LN prototype provides support mainly for selection operators.

The lower layer -*DADT/LN runtime* - provides support for interfacing DADT runtime and LN runtime. This is achived by using instances of the *LNNodeManager* on each sensor device and *LNDistrNodeManager* instance on the controller node.

Communication between sensor devices and controller node is handled by LN runtime layer and from the application's view achieved by use of *RequestMsg* and *ReplyMsg* messages. Classes *SelectAll*, *SelectAny* implement DADT selection operators and allow to invoke respective send methods provided by LN API.

4.2.4 Implementation Details

This section attempts to explain the operation of the DADT/LN prototype developed as part of this work by considering the sequence of method calls made during execution. The operation of the simulation platform as well as the nodes itself is abstracted from the explanations that follow, as is the actual *.dadt* syntax used by the application programmer himself to trigger these operations.

4.2.4.1 The DADT/LN prototype on the Controller

Figure 4.4 presents the operation of the DADT/LN prototype on the controller (which is typically PC-based). As shown in the figure, the implementation running on each sensor node consists of the following entities:

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

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

A *DADT Instance* allows for collective access to multiple ADT instances, as it was described in Section 4.1.2.

An *Expression Tree* is a special object that allows to build an abstract tree for the DADT View, based on the set of DADT Properties.

A *DADT 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 3.2.4.2.

The instantiation of a DADT type by the application programmer's code[4] causes the following actions to take place at the Controller:

- the Client Node creates an instance of the DADT type and use it to perform collective operations on the network.

[4]This is written in the DADT specification language.

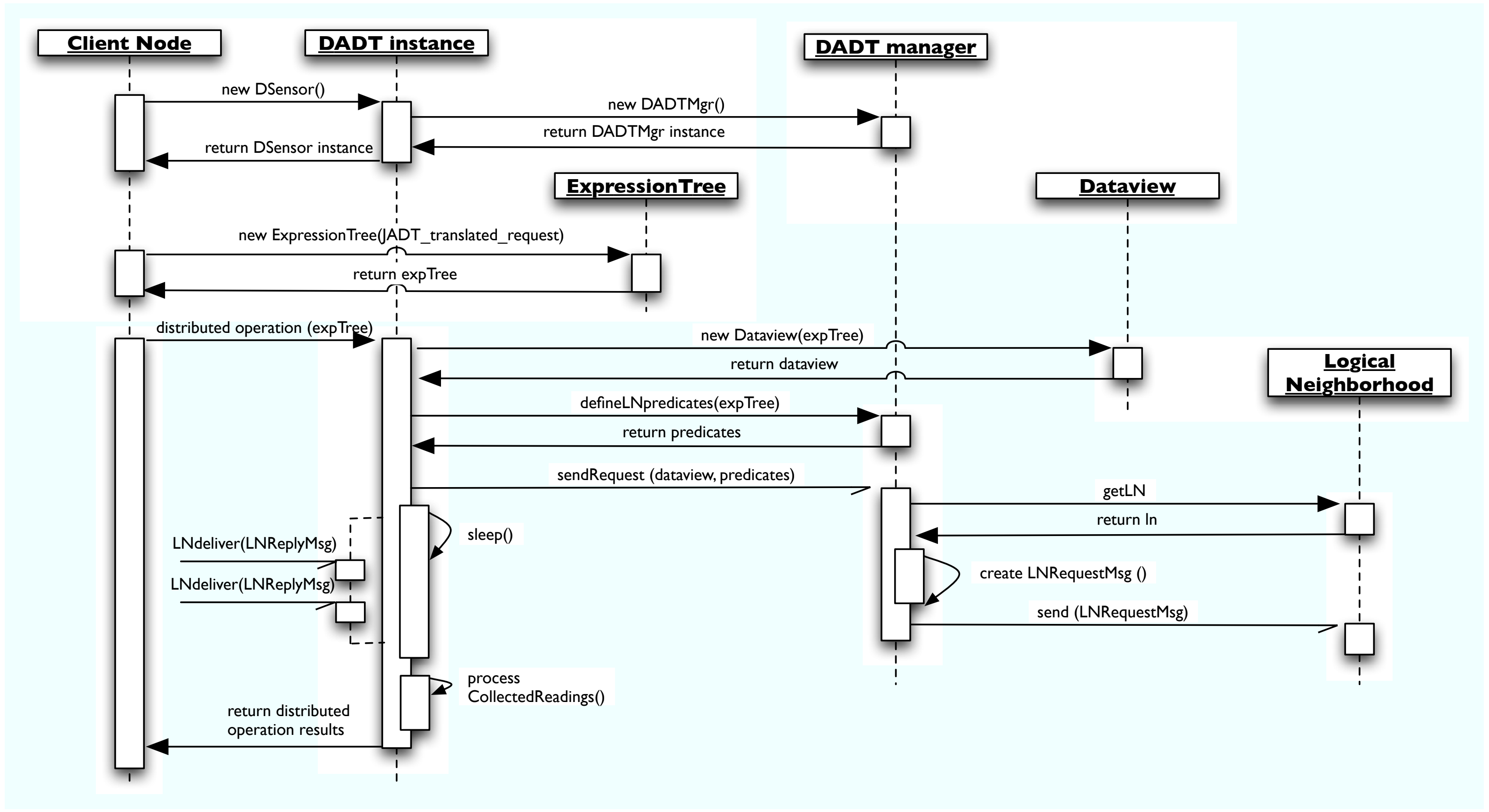

Figure 4.4: Operation of the DADT/LN prototype on Controller

- a new expression tree is created to provide a representation of DADT View defined by an application programmer.
- the DADT instance creates an instance of the DADT Manager.

When the application programmer's code requests the execution of a distributed operation on the WSN, the Client Node forwards the request to the DADT instance, which processes the request and communicates with an underlying WSN.

The DADT instance performs 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,
- subsequently uses the DADT Manager to specify LN for the provided request, by using a special procedure of mapping DADT properties from the expression tree object into LN predicates,
- using the DADT manager it constructs and sends a request message to underlying WSN, and
- being a separate thread, it sleeps until, if required, the result of the request is received from the network layer.

If no reply from the WSN nodes is expected the Client Node continues its work in the normal mode. This situation could be described by an example of the request contatining 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. Further it invokes the relevant methods to collect and process the received readings. Finally, The DADT instance returns the result of the DADT request to the Client Node.

4.2.4.2 The DADT/LN prototype on the sensor device

Figure 4.5 presents the operation of the DADT/LN prototype on the sensor device (which may be either simulated or a real node).

Later in 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 abstraction of all of the sensors within the device, and this abstraction resides on the device.

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

- *Sensor Node*
- *Sensor ADT instance*
- *ADT 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 4.1.1.

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

An *ADT 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.

Intialisation of the Sensor ADT instance is performed possibly multiple times on a given Sensor Node, as a node might consist of multiple sesnsors. Following this, the sensor ADT instances are bound to a particular DADT type by calling the ADT manager[5].

When the lower layer (which runs the LN algorithm) delivers a message to the Sensor Node, the ADT Manager is used to processed the request message.The request message contains details about a DADT View (see Section 3.2.4.2) which is used later to filter from the sensor ADT instances on the given Sensor node those that fit into the requested DADT View.

If 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, as the

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

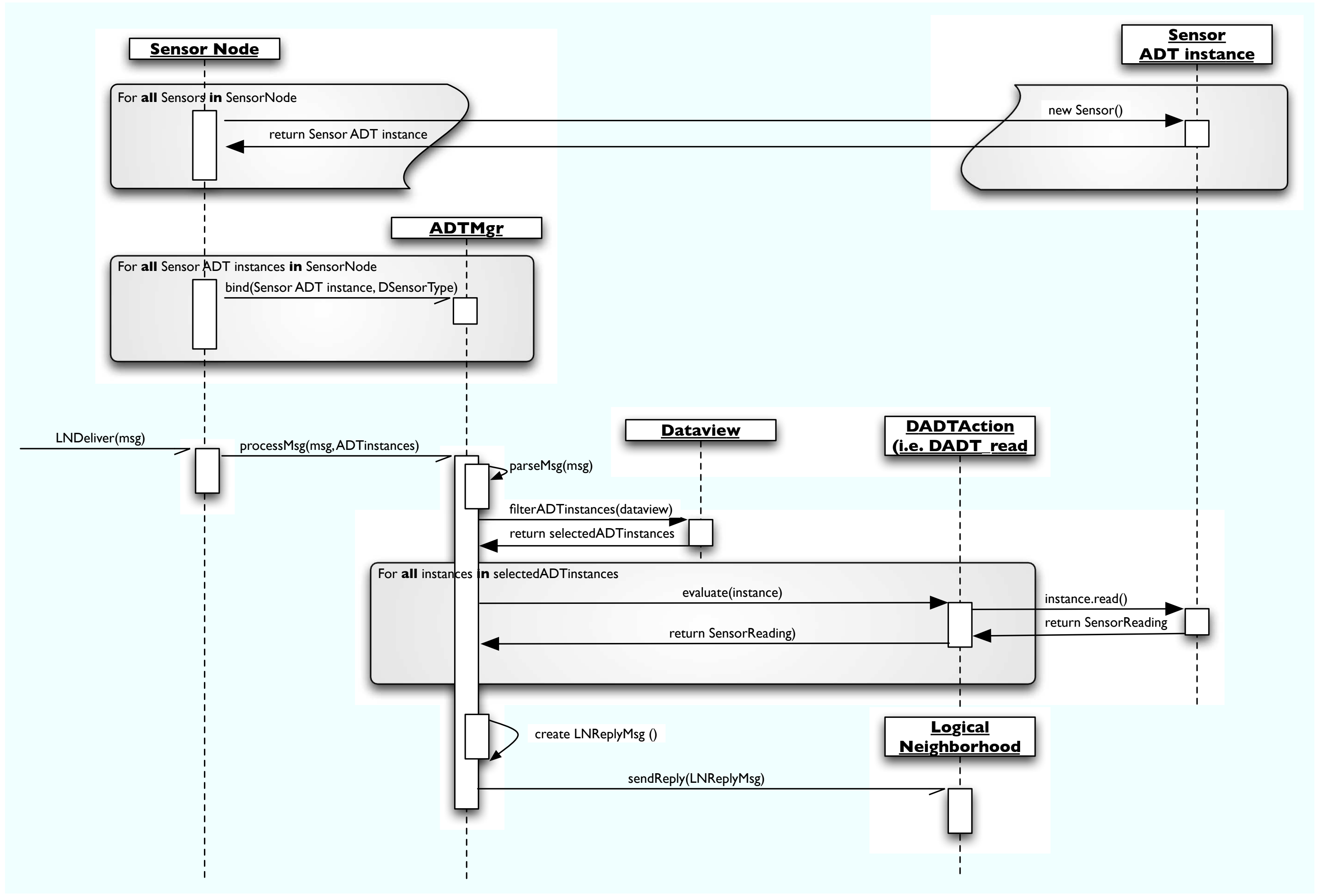
Sensor Node
Sensor ADT instance
For all Sensors in SensorNode
new Sensor()
return Sensor ADT instance
ADTMgr
For all Sensor ADT instances in SensorNode
bind(Sensor ADT instance, DSensorType)
LNDeliver(msg)
processMsg(msg,ADTinstances)
parseMsg(msg)
Dataview
DADTAction (i.e. DADT_read
filterADTinstances(dataview)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

Figure 4.5: Operation of the DADT/LN prototype on sensor device

DADT view is expressed in the form of the LN predicates. 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.

The request message also contains information about the DADT action to be performed on device (see Section 3.2.4.1). The ADT 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 it can be seen on the bottom right section of Figure 4.5.

4.2.5 The DADT/LN prototype in the simulated environment

4.2.5.1 JiST/SWANS

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 a particular discrete event simulator called JiST, and the SWANS network simulator built on top of JiST.

4.2.5.1.1 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 [**?**]:

- *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.

4.2.5.1.2 Java In Simulation Time (JiST) JiST [?] 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[6], as it can be seen in Figure 4.6.

The execution of a JiST program can be understood by considering example as shown in Listing 4.10

```
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 4.10: Example JiST program (reproduced from [?]

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

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

Listing 4.11: 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 JistAPI functions used in the example code are used to perform the code transformations. The method call to myEvent is now scheduled and executed by

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

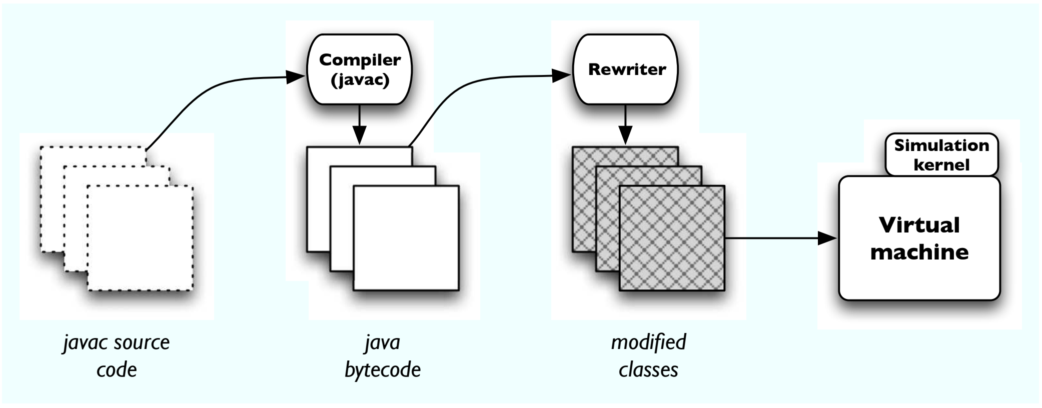

Figure 4.6: The JiST system architecture (reproduced from [?])

the simulator in simulation time. Simulation time differs from "actual" time in that the advancement of actual time is independent of application execution.

4.2.5.1.3 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 [?], and is built upon the JiST discrete event simulator described in Section 4.2.5.1.2. It is organised a a collection of independent, relatively simple, event driven components that are encapsulated as JiST entities.

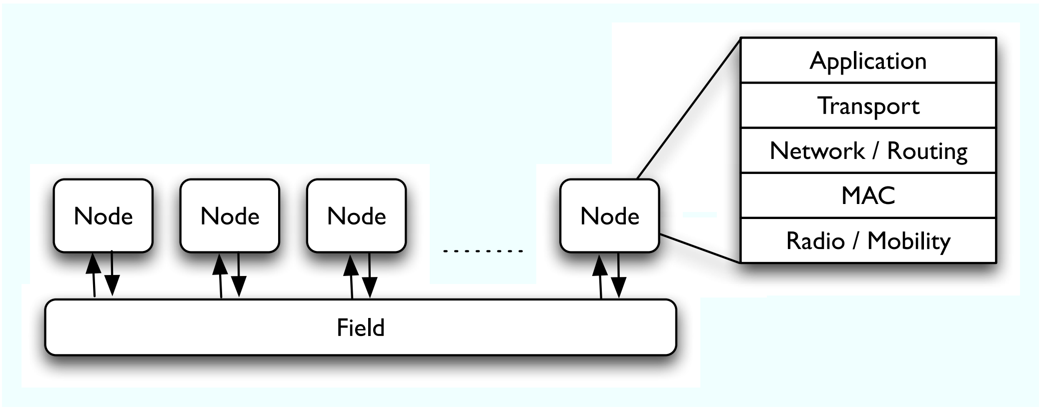

Figure 4.7: SWANS architecture

SWANS has the following capabilities [**?**]:

- The use of interchangeable components enables the construction of a protocol stack for the network, and facilitates parallelism, and execution in a 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 4.7.

4.2.5.2 Simulation using JiST/SWANS

The DADT/LN prototype was tested on the SWANS WSN simulator, that is built upon the JiST discrete event simulator (see Section 4.2.5.1.2).

The DADT/LN prototype code is wrapped in the JiST API, and is loaded on to a simulated node. As described in the previous section, there are two kinds of nodes in the DADT/LN prototype:

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

The Controller Node was implemented as a separate node on the JiST/SWANS simulator, and was run as an undependet WSN simulated node for the purposes of simulation. The Sensor Device implementation (see Section 4.2.4.2) was loaded on to all but one of the nodes in the simulator.

The simulation was run on networks of upto 50 nodes to empirically verify the robustness of the work done as part of this thesis.

4.2.6 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 [**?**].

The Controller application was executed as a host application on the host machine (a PC), while the other Sun SPOTs ran the Sensor Device implementation as on-SPOT applications (see Section 3.4.4 for a description of host and on-SPOT applications).

Deployment of the DADT/LN prototype on Sun SPOT devices involved few challenges. Whereas JiST/SWANS simulator allows to run unmodified Java code on the simulated node, Squawk JVM on Sun SPOTs introduces severe limitations towards the code. Application code has to be compliant with CLDC 1.1 specification of Java ME, and therefore lacks support for a number of APIs.

Chapter 5

Evaluation, Conclusions, and Future Work

5.1 Evaluation

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

- Packet processing workload on the application layer.
- Ease of implementation.

The first metric was used to compare the performance of the DADT/LN prototype against that of the original DADT prototype used as the basis for this work [**?**]. In the original DADT prototype, a request message was replied to or discarded in the application layer on the basis of the evaluation of an abstract tree representing specified scope of the operation. In the implementation presented in this work, the integration with the LN approach results in unsuitable request messages being discarded on the basis of predicate evaluation in the network layer.

A series of simulations were run using the JiST/SWANS simulations to determine the number of request messages discarded at the network layer and the number passed on to the application layer by the LN predicate matching algorithm. The sum provides the total number of packets processed in the application layer of the original DADT prototype.

It was found that the number of messages processed in the application layer was lower in the implementation produced as part of this work.

5.2 Conclusions

5.3 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.