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

*Galiia Khasanova*

Master of Science
Dipartimento di Ingeneria e Scienza dell'Informazione
University of Trento, Italy
2008

# Abstract

This thesis presents blah blah blah.

# 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 |
| OS | Operating System |
| 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

Jo: quote is fine, but better Sth Like : Mark Weiser once wrote in his visionary paper from whenever [22]:

> *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)

Recent progress in micro system technologies and wireless communication made it possible to deploy wireless sensor networks (WSNs) that contain a large number of sensor nodes, multifunctional devices characterised by their low- cost, low-power, and small form factor, that can communicate across short distances using Radio Frequency (RF) communication [3].

WSNs are being used in a wide range of applications in military and civilian operations such as, for instance, health monitoring, environment monitoring, data acquisition in dangerous environments, and target tracking.

# Chapter 2

# Introduction to Wireless Sensor Networks

This chapter begins with an introduction to wireless sensor nodes. This is followed by a presentation of Wireless Sensor Networks (WSNs) and discussion of their main features.This is further followed by an introduction of the WSN protocol stack. The chapter concludes with a presentation of the programming models for WSNs and benefits of their use.

Wireless Sensor Networks (WSNs) are typically deployed randomly in a possibly large area where phenomena are required to be monitored, and consist of the large number of the light-weighted and tiny devices (sensor nodes) that can be easily deployed.

## 2.1 Sensor Nodes

Typically, a sensor node consists of the following elements (as it can be seen from the Figure 2.1)

- *Sensing unit*, which is comprised of a number of sensors and analog-to-digital converters.
- *Tranceiver*, which facilitates node-node communication using a variety of

techniques.

- *Processing unit*, that usually 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.

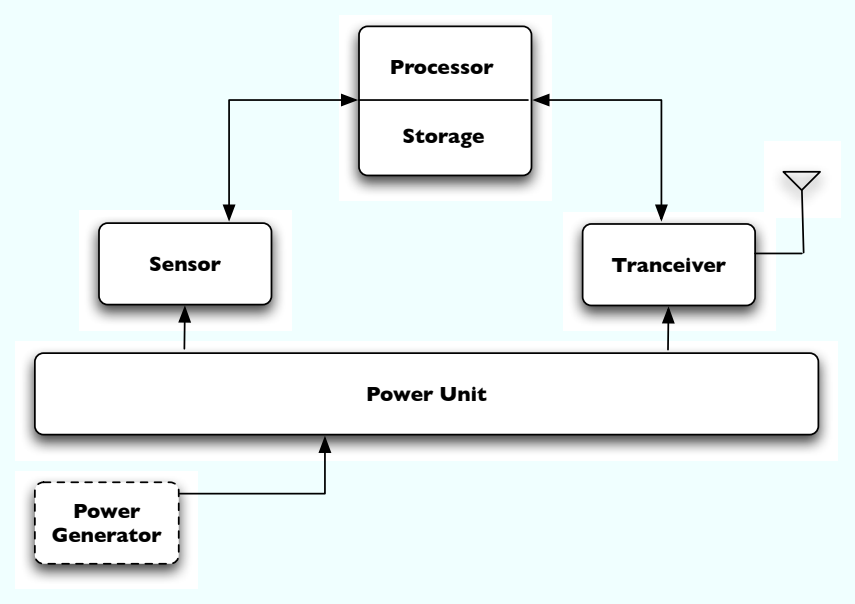


Figure 2.1: Architecture of a sensor node (adapted from [3]).

Due to the small size of the devices, sensor nodes have a number of constraints which affect the WSN built on top of it, among those are [24]:

- *Power consumption constraint* due to the fact that sensor nodes have limited energy supply, therefore energy conservation is the main concern when WSN application are implemented.
- *Computation restriction* which is caused by the limited memory capacity and processing power available on the sensor node, thus possibilities to use data processing algorithms on a node are seriously limited.

- *Communication constraint* is caused by the minimal bandwidth and a limited quality of service provided by the sensor node's hardware.

Additionally, the deployment of sensor nodes in the WSNs should be cost-effective, and therefore 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 that was presented in [3] is adapted generic protocol stack [4]. As it can be seen from the 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 is resided.

## 2.3 Routing in WSNs

we need sth here

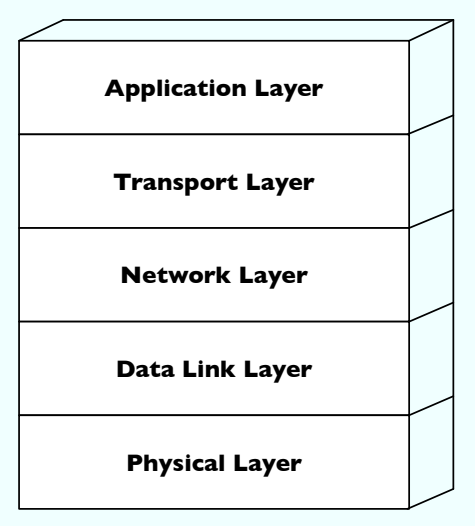


Figure 2.2: WSN protocol stack (reproduced from [3])

# 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 contibues with 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) [18], 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) [22] - 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 appli-

cations 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 [19].

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 [19].

According to Yu et al [25], 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 [11] (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 [13], Cougar [7], SOS [12], Agilla [8].

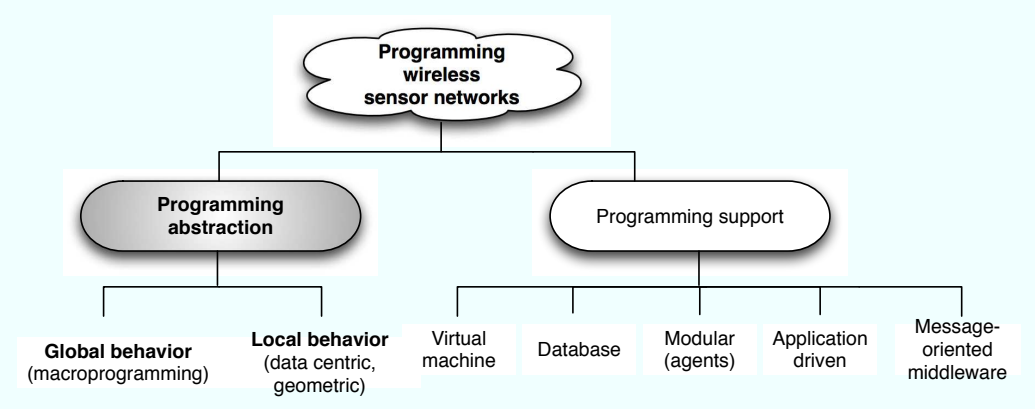


Figure 3.1: Taxonomy of WSN programming models (reproduced from [11])

- *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 [14], Kairos [9] and EnviroTrack [2].

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* [11].

In the former case, the sensor network is programmed as a whole, and gets rid of the notion of individual nodes [19]. Examples of macroprogramming solutions include *TinyDB* [15] and *Kairos* [9].

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.

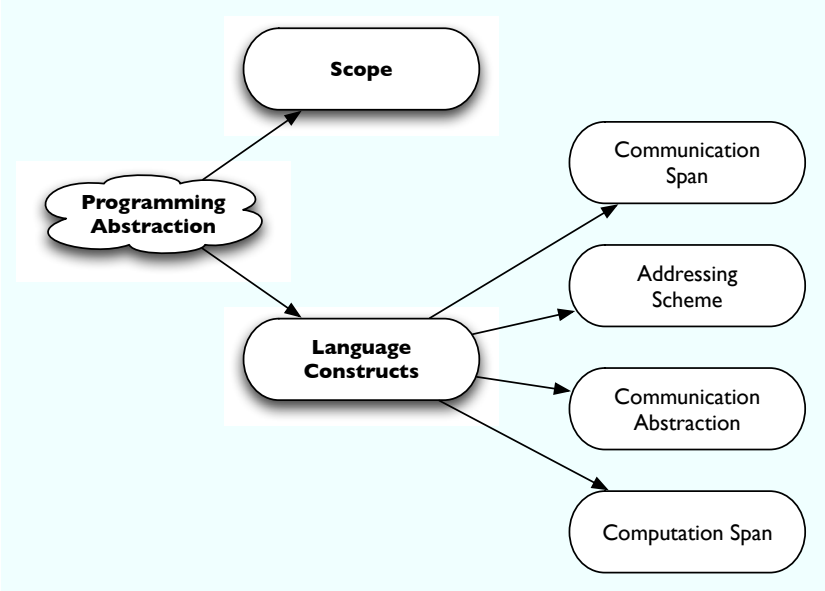


Figure 3.2: Classification of Programming Abstractions

Programming abstractions may also be classified on the basis of the nature of the language constructs made available to the WSN programmer [19]. Some of the metrics used for classification are[1]:

- 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 [19] 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.

[1]Only relevant to this work classification bases were selected from [19] for discussion

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 node(s) 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.

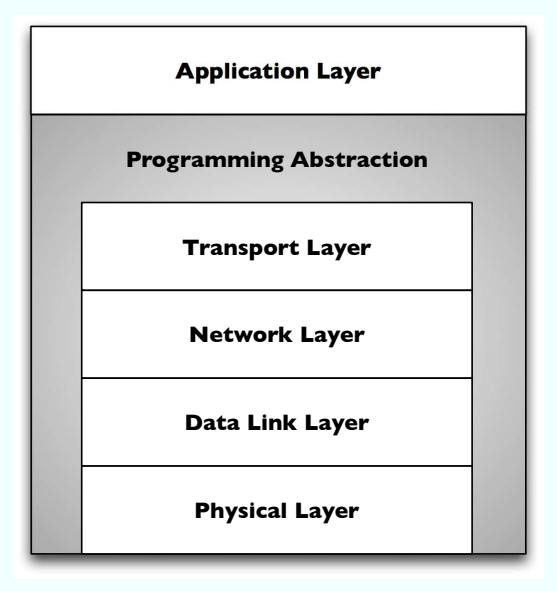


Figure 3.3: Programming models on the WSN protocol stack (adapted from [19])

### 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 [20]. The rest of this section discusses the concepts and provides the reader with the relevant background[2].

### 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 [1]. 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 [10]. 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 already sucessfully used in the different areas of science. The rest of this section focuses on the application and extension of ADT concept to be used in WSNs.

### 3.2.2 ADTs in WSNs

A WSN, as defined earlier, consists of number of sensor nodes. Each sensor node may include several sensors. By defining each sensor as an ADT instance, the concept of ADTs can be used in WSNs, as this is shown on the Figure 3.4.

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 peresented by *Sensor Node*, and be later used by the application developer[3].

[2]Code snippets provided as examples are were described in [20]

[3]Further details about the ADT specification and intantiation is provided in the Section 4.1.1

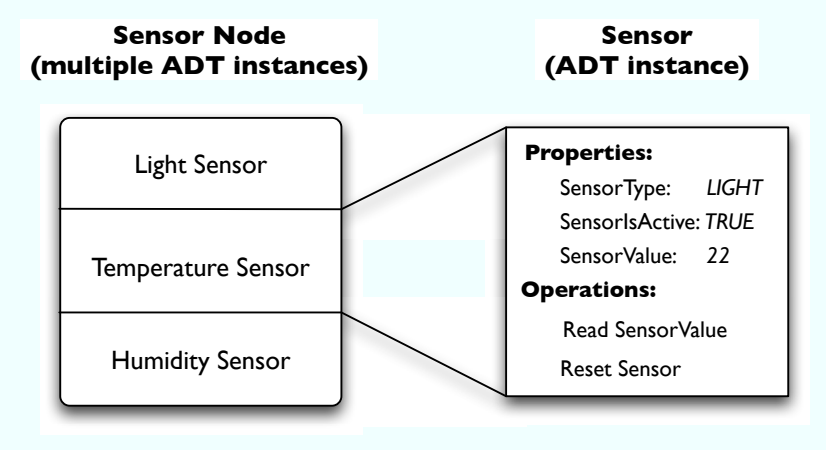


Figure 3.4: Abstraction of sensor node through multiple ADTs

### 3.2.3 Data and space ADTs

As a WSN consists of a collection of sensor nodes that are distributed in space, ADTs additionally can also express the spatial location of a given sensor node. Thus, ADTs used in WSNs can 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" [20]. 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 [20].

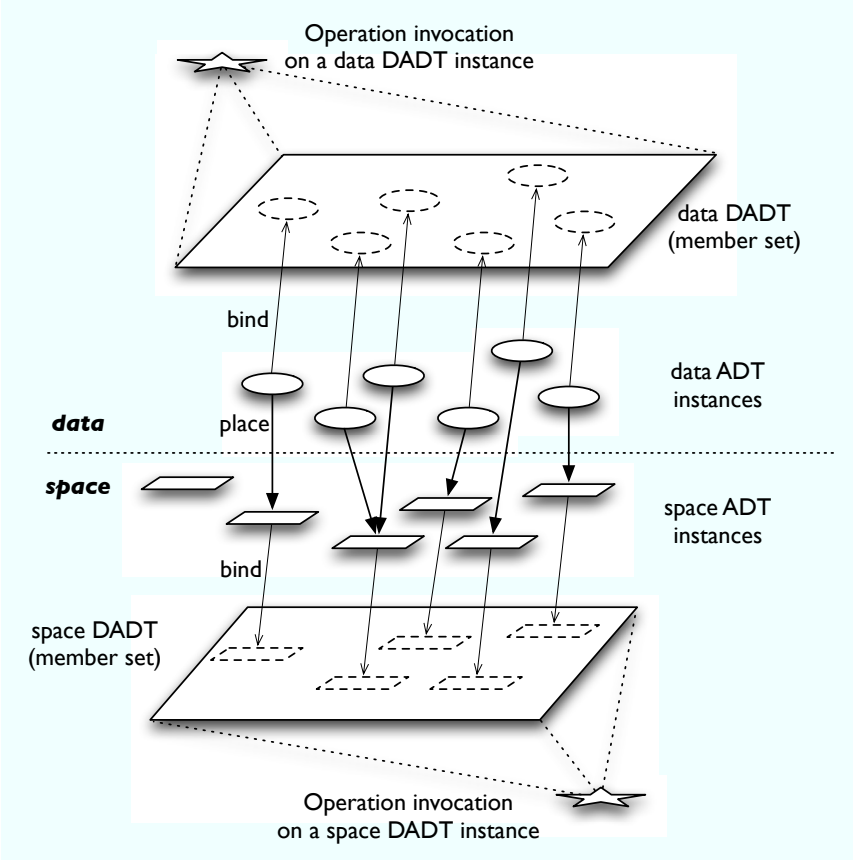


Figure 3.5: Data and space in the DADT model (reproduced from [20])

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*, or *Selectors*, that allow to perform distributed operation on a subset of the instances in the member set.
- *Conditional Operators* that make the code in the DADT method dependent on a global condition on the member set.
- *Iteration Operators* that allow iterating over the ADT instances in the target set, and thus permit access to individual ADT instances.

DADT actions are special DADT programming constructs. They are defined 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* 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 [20]. DADT view may either be:

- Data View, or
- Space View.

The concept of DADT Views is summarised in Figure 3.6.

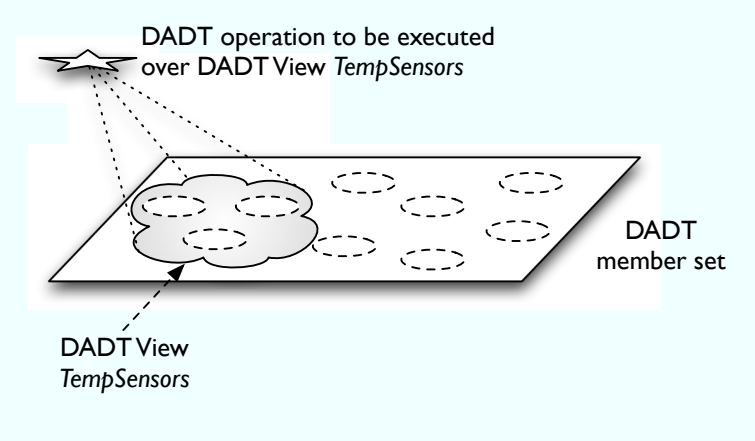


Figure 3.6: DADT views (reproduced from [20])

## 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, because 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. This might lead to an increased complexity of the code [17].

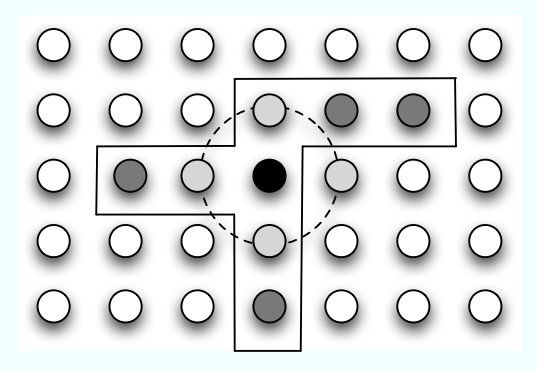

Figure 3.7: Representation of logical and physical neighborhood of a given node. *The dashed circle represents the physical neighborhood, whereas the solid polygon represents (one of) the node's logical neighborhood* (reproduced from [17])

Mottola and Picco [17] 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 example above, 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 [17, 18][4], 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 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

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

```
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 [17]. 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 updation of node profiles. The routing mechanism can be divided into two phases [16]:

### 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 sends 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 determined during the first phase. This path is called a *decreasing path*.

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 [22].

On the other hand, it is widely accepted that development times are greatly reduced upon the use of managed runtime languages such as Java [22]. Therefore, currently prevalent WSN programming practice trades developer efficiency for memory efficiency.

However, Simon et al [22] state the benefits accrued 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.

### 3.4.1 The Sun SPOT hardware platform

Sun Microsystems has, on the basis of the arguments discussed in the previous section, 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.

Figure 3.8: Sun SPOT device

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 (formerly ChipCon) 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 [22]:

- 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. This reduces memory consumption.
- It allows for inter-device application migration.

- It supports 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 (See Figure 3.9.

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 [22], 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” [22].
- 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: [23]:

- On-SPOT applications
- On-Host applications

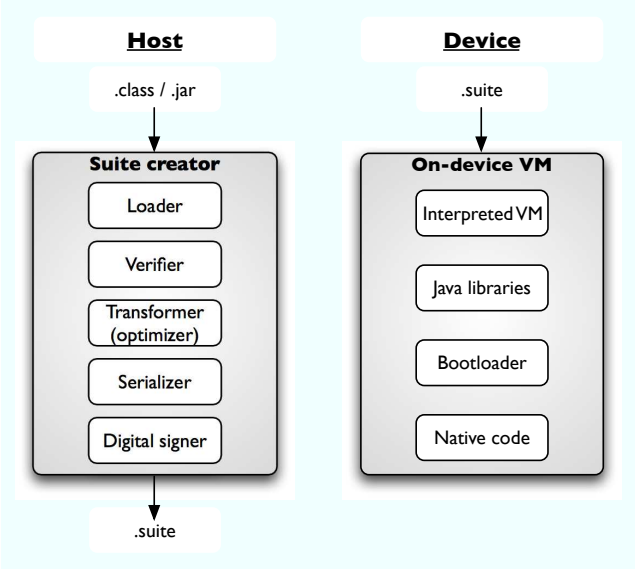


Figure 3.9: The Squawk Split VM Architecture (reproduced from [22])

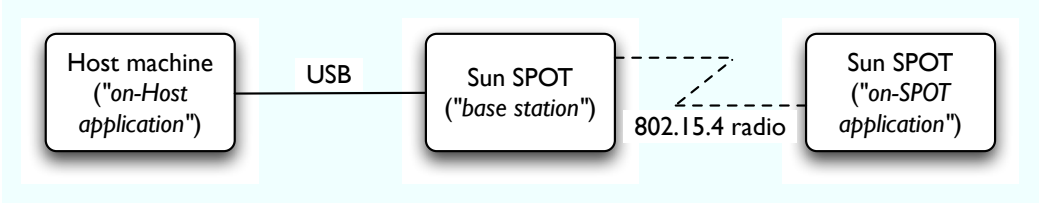


Figure 3.10: Types of Sun SPOT applications (adapted from [23])

*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

re-write! probably the sequence is outdated

This chapter begins with implementation details underlying Distributed Abstract Data Types (DADTs), a short description of existing DADT prototype [20], and an outline of the limitations of the current prototype. The chapter then provides information about author's contribution and discusses architecture of implemented DADT/LN prototype. This is followed by the implementation details the DADT/LN prototype which allowed use of DADT concepts in the real-world WSNs. This is folowed by a presentation of the simulation environment, simulating tools that were used for verification of the implementation. This chapter further presents the evaluation results for the DADT/LN prototype run on the Sun SPOTs. The chapter concludes with discussion of possible extensions of the prototype.

## 4.1 The DADT Prototype

Use of DADTs can be applied to the context-aware distributed applications, as it was discussed in the Section 3.2. This section provides reader with further details of DADT concepts, a presentation of the existing DADT prototype [20]

and discusses its limitations.

### 4.1.1 ADTs specification and instantiation

As it was mentioned in the Section 3.2.2, each sensor node in WSNs can be abstracted though a multiple number of ADT instances (see Figure 3.4) and therefire hide the details of sensor node abstraction from the application developer. Code snippets below are provided to present this idea in further detail.

The specification for such ADTs[1] could be defined as Java interface as presented in the 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 values that holds the sensor reading.
- A boolean value that stores information about sensor's state of activity

and the operations *read* sensor readings and *reset* sensor.

[1] According to our classification in the Section 3.2.3 this is an example of the Data ADT

Therefore, 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 the 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 introduced in the Section 3.2 and will be extended with examples of DADT specifications and instances in this chapter.

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{
  // type of ADTs
    Class distributes = Sensor.class;
  // distributed operations:
        void resetAll();
        double average();
}
```

Listing 4.3: Data DADT specification (reproduced from [20])

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

- *resetAll()* that 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).

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

DADT specifications can be instantiated as an object of a class, and can be used to perform allowed distributed operations mentioned above (see Listing 4.4).

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

Listing 4.4: DADT Instantiation (adapted from [20])

#### 4.1.2.1 Binding

As it was specified in the Section 3.2.4, a DADT member set of is being built by the set of ADTs that are available for collective access, in this case, 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 as shown in Listing 4.5, where the Sensor ADT (see Listing 4.1) is bound to the DADT type *DSensor* defined in Listing 4.4.

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

Listing 4.5: Binding ADT instances to a DADT instance

#### 4.1.2.2 DADT Operators and Actions

The Section 3.2.4.1 has already introduced the definitions of DADT Operators and DADT Actions, hence, in this chapter the implementation details will be presented.

Use of DADT selection operator *all* can be explained through the extended example of DADT Specification (see Listing **??**) which uses pseudolanguage. This code snippet is shown in the Listing 4.6.

```
class DSensor {
  ...
  public void resetAll(){
         (all in targetset).reset();
  }
}
```

Listing 4.6: Defining a DADT action

Distributed operation *resetAll* exploits selector *all* in order to get an access to all ADT instances of the DADT target set, and subsequently invokes aperation *reset* on the ADT instance, as it was declared by the ADT specification (see Listing 4.1).

rewrite here

To perform a DADT read operation, an action is defined as shown in Listing 4.7.

```
public class DSensor_read_Action implements DADT.Action{

  public Object evaluate(Object ADTInstance){
        Sensor localSensor = (Sensor) ADTInstance;
    return localSensor.read();
  }
}
```

Listing 4.7: Defining a DADT action

This performs the operation on the ADT Instance of type *Sensor* (see Listing 4.1).

When the application programmer attempts to execute a DADT operation, such as for instance, the computation of the average across all sensors in its target set (see Listing 4.4), actions of the sort defined above are called[2]

[2]The details of the implementation of actions are outlined in Chapter 4.

#### 4.1.2.3 DADT Views

DADT Views an effective tool for the application developer to define scope of the distributed operation. As it was mentioned in the Section 3.2.4.2, the concept of DADT Views is based on the definition of *DADT Property*.

To continue on the example running throughout this section, If the application programmer wished to partition the set of sensors bound to the data DADT type *DSensor* (See Listing **??**) to refer to only those data ADT instances that are temperature sensors, a data view can be delared as shown in the Listing 4.8

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

Listing 4.8: Definition of DADT Data View

The DADT name *DSensor* in this case refers to its member set, and the data view *temperature* is defined as a subset of this member set and contains only sensor nodes with temperature sensors for which evaluation of the property *isActive* returns *true*. The data view definitions are used to restrict the scope of the DADT operations.

### 4.1.3 Implementation Details and Limitations

[20] presented a prototype that enabled the use of DADTs to facilitate distributed application programming.

DADT prototype *JADT* supports is a Java-based application development and consists of two parts:

- *Transltor*, that proviides translation of Java program extended with DADT programming constructs into conventional Java classes.
- *Run-time* library is used at translation step and allows to run created Java classes in JVM.

Run-time library 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 IP Multicast, and it allows to deliver information to the ADTs bound to a specific DADT.

The DADT prototype has presented a proof of DADT concept. While this approach is clearly applicable to WSNs, the prototype itself did not support WSN abstractions, and there were several limitations to it:

- The lack of a routing mechanism.
- Limitations in portability to real WSN nodes.

## 4.2 The DADT/LN Prototype

### 4.2.1 Motivation

As mentioned in the previous section, the DADT prototype [20] showed that DADT approach theoretically can be used for distributed WSN applications, but existing limitations prevent its use in WSN simulators or real nodes.

This work makes the following contributions:

- Enhance 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 [16] to enable abstracted communication between groups of nodes in the WSN defined by DADTs.
- Verification of the utility of DADT abstractions in the WSN application layer.

### 4.2.2 Architecture

### 4.2.3 Overview

The overview of the workflow involved in using DADTs to enable WSN application programming is as shown in Figure 4.1. The user writes application layer

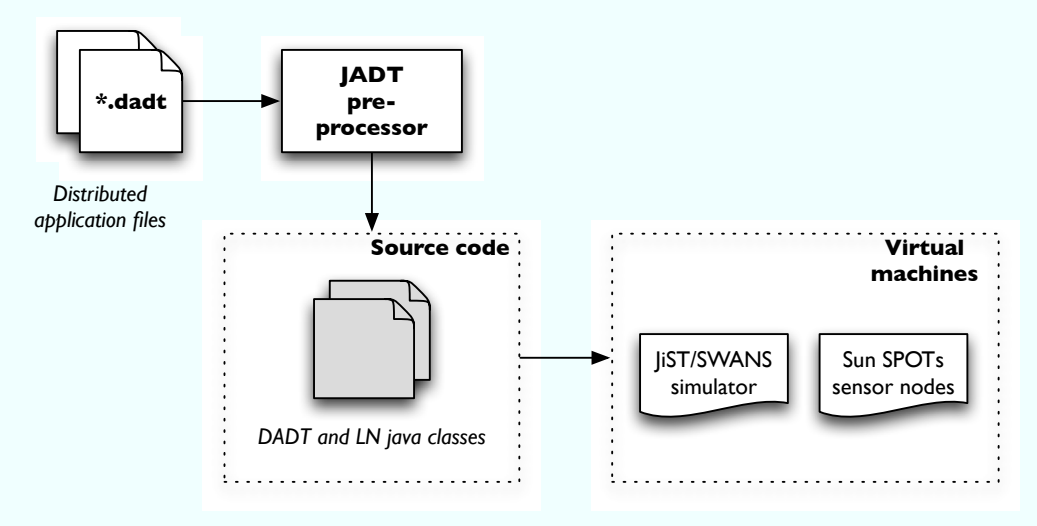


Figure 4.1: Workflow for development of an application that uses the DADT/LN prototype

code for the WSN using a DADT language in a series of *.dadt* files. A 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 [20]). In order to facilitate routing to LNs defined by the use of DADT views, the DADT infrastructure is interfaced with the a previously developed implementation of LNs.

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

- the JiST/SWANS simulator [6, 5]. See Section 4.2.5.1 for details on the implementation of the aforementioned simulator
- a collection of Sun SPOT wireless sensor devices [22] (see Section 3.4) to execute the application on real sensor nodes.

#### 4.2.3.1 Explanation of terms used

This section explains the terms used in the DADT/LN prototype developed as part of this work. The WSN network in this prototype consists of two types of devices

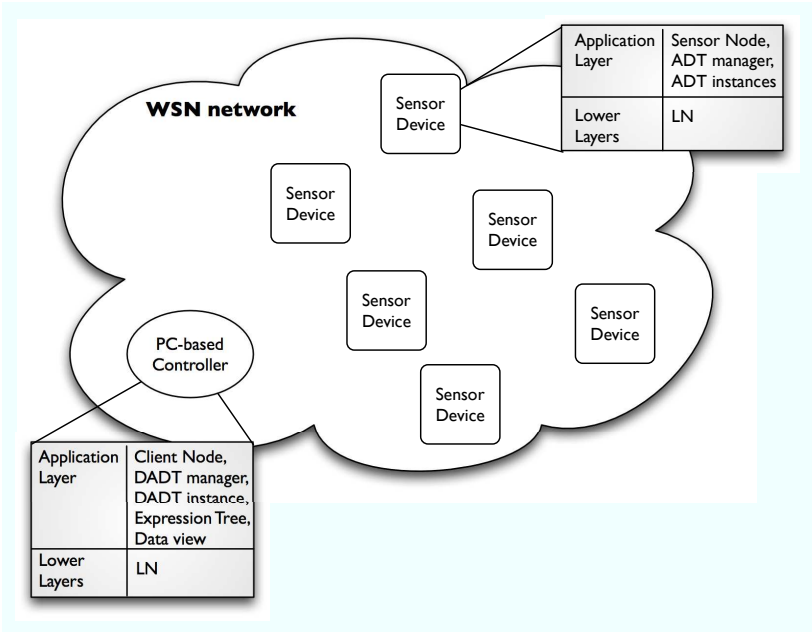


Figure 4.2: Schematic Representation of the WSN as abstracted in the DADT/LN prototype

as shown in Figure 4.2:

- *Controller:* Typically PC-based, this node contains in the application layer the client node, the expression tree, the DADT instance, the DADT manager; the LN implementation is used in the network layer. The user's application code resides on the controller node.
- *Sensor Device:* This is a sensor device such as a Sun SPOT, and holds the following entities:
  - *Application Layer:* It includes a Sensor Node that in turn consists of multiple sensor ADT instances, and an ADT manager.
  - *Network Layer:* The LN implementation is used.

### 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 **??** 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:* A Client Node is an abstraction that consists of a DADT instance. The application programmer's requests to the network are issued by the Client Node.
- *DADT Instance:* A DADT Instance is explained in Section 4.1.2, and allows for collective access to multiple ADT instances.
- *Expression Tree:* An expression tree is a specific construct that is to build a hierarchical data view.
- *DADT Manager:* The 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.
- *Data View:* Data views are explained in Section 3.2.4.2, and is a mechanism for partitioning the collection of ADT instances bound to a particular DADT type.

The instantiation of a DADT type by the application programmer's code[3] causes the following actions to take place at the Controller:

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

- The Client Node creates an instance of the DADT type that is used to perform collective operations on the network.
- A new expression tree is created to provide a hierarchical representation of the application programmer's definition of a data view.
- 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 following actions take place:

- The Client Node forwards the request to the DADT instance. The DADT instance:
  - creates a Data view using the expression tree (see above),
  - subsequently uses the DADT manager to construct LN predicates from the expression,
  - sends a request message to the network using the DADT manager, and
  - sleeps until the result of the computation is received from the network layer *(N.B.: The DADT instance is implemented as a separate thread)*.
- When the result of the distributed computation is received, the Client Node is notified. The Client Node then executes the appropriate function on the DADT instance to collect and process the readings. The DADT instance returns the processed readings to the Client Node.

#### 4.2.4.2 The DADT/LN prototype on the sensor device

Figure **??** presents the operation of the DADT/LN prototype on the sensor device (which may be either simulated or a real device). As shown in the figure, the implementation running on each sensor node consists of the following entities:

- *Sensor Node:* 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]

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

- *ADT Manager:* The 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.

Sensor ADT instance intialisation 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 a DADT Data view (see Section 3.2.4.2) which is used to filter from the sensor ADT instances on the given Sensor node those that fit into the Data view.

*N.B.: 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 dataview, as the data view is expressed in the form of an LN predicate. 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 a description of 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 Data view.

[4]The term *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. This abstraction resides on the device.

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

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 **??**.

### 4.2.5 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 [21]:

- *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 [6] 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 4.5.

[6]N.B.: The bytecode rewriter and the simulation kernel are both written in Java

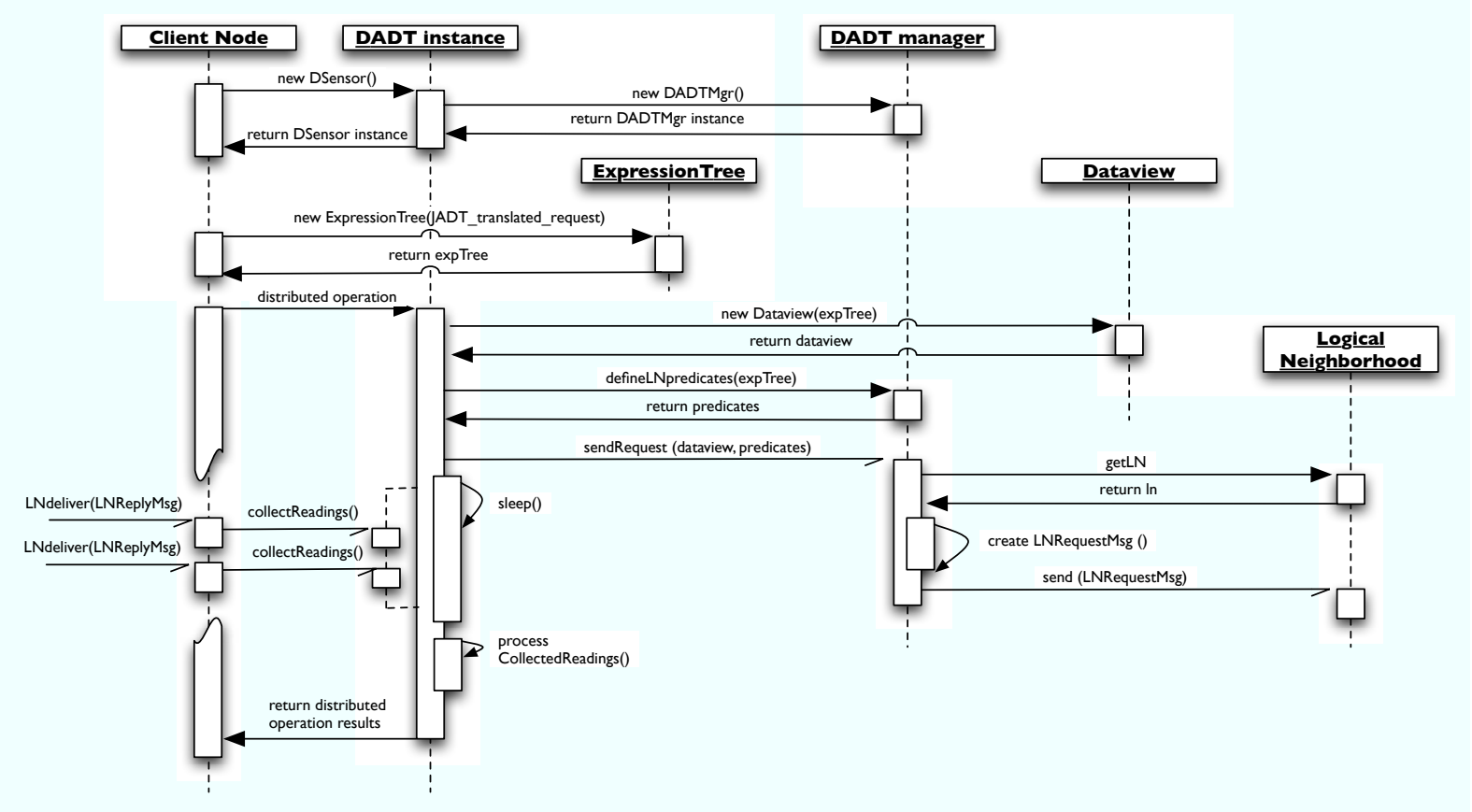


Figure 4.3: Operation of the DADT/LN prototype on Controller

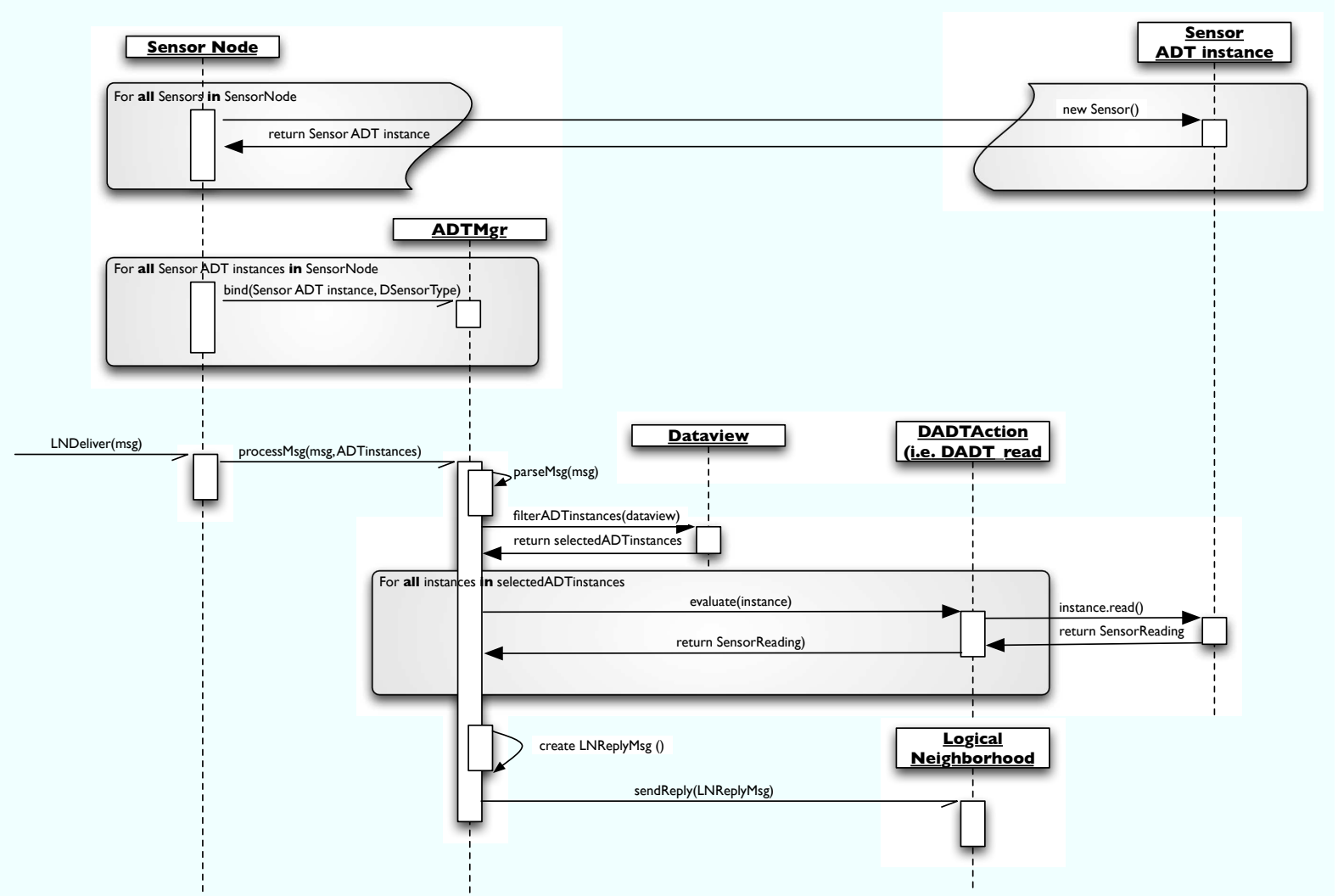


Figure 4.4: Operation of the DADT/LN prototype on sensor device

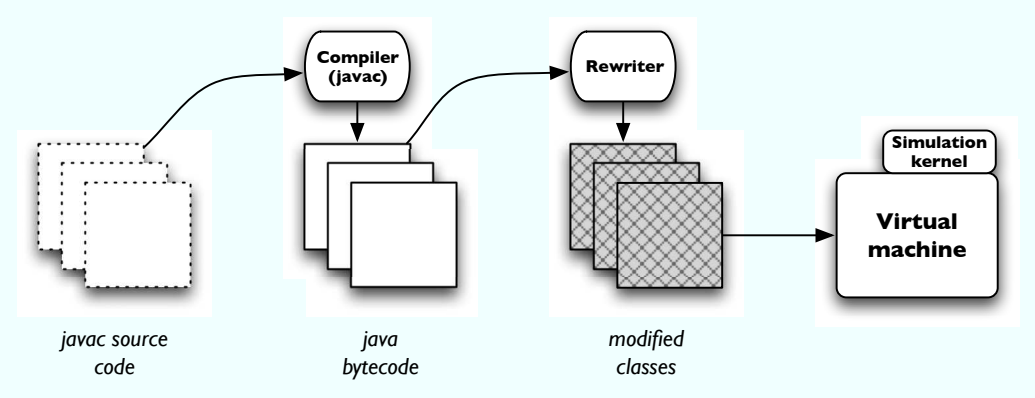


Figure 4.5: The JiST system architecture (reproduced from [6])

The execution of a JiST program can be understood by considering example as shown in Listing 4.9

```
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 4.9: Example JiST program (reproduced from [6]

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 4.10: 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 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 [5], 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.

SWANS has the following capabilities [5]:

- 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 4.6.

### 4.2.5.2 Simulation using JiST/SWANS

The DADT/LN prototype was tested on the SWANS WSN simulator, that is buit 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:

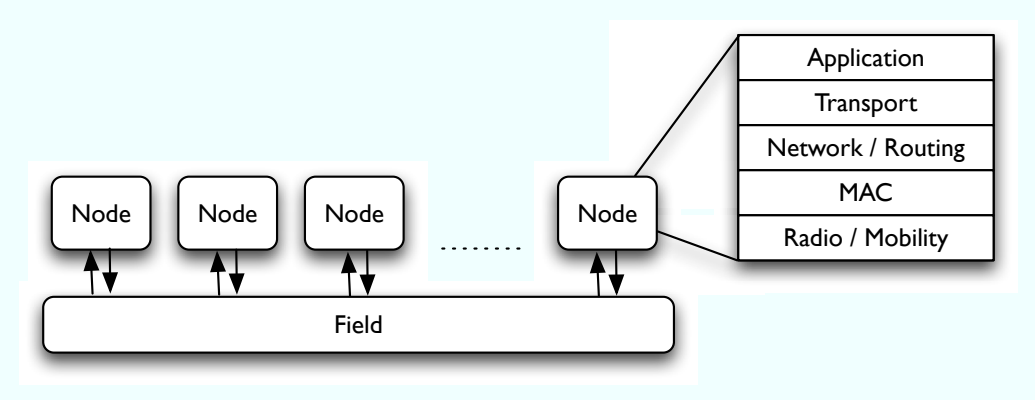


Figure 4.6: SWANS architecture

- *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 assumed not to be PC-based 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 [22].

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).

## 4.3 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 [20]. In the original DADT prototype, a request message was replied to or discarded on the basis of an expression tree evaluation in the application layer. 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.

The second metric used the number of lines of code required to implement a simple distributed averaging application. While not an ideal metric, it quantifies the difference in the order of magnitude of coding effort required on the part of the application programmer to produce WSN applications with and without the use of programming abstractions. To this end, a simple WSN application was written on the Sun SPOTs to calculate the distributed average.

The difference in the number of lines was **.

## 4.4 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 due to limitations in the current LN implementation.
- *Extending support for Space DADTs:* Currently, the prototype does not use the notion of space. Therefore, a possible avenue for future work could include the implementation of Space DADTs, and the definition of Space views that are analogous to Data views.
- *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.

## 4.5 Conclusions

# 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] BARR, R. SWANS-Scalable Wireless Ad hoc Network Simulator Users Guide, 2004.

[6] 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).

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

[8] 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.

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

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

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

[12] 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).

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

[14] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence* (2005).

[15] 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).

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

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

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