Distributed Programming Abstraction for Java-based wireless sensor nodes (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.

Acknowledgements

Bla

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*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Defintions and Acronyms

ADT	Abstract Data Type
DADT	Distributed Abstract Data Type
JiST	Java in Simulation Time
LN	Logical Neighborhoods
MAC	Media Access Control
RF	Radio Frequency
SPOT	Sun Small Programable Object Technology
SWANS	Scalable Wireless Ad hoc Network Simulator
WSN	Wireless Sensor Network

Chapter 2

Background

"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 [19]

This chapter presents a brief discussion of the concepts, algorithms, and simulation and hardware platforms that were used during the course of this work. The chapter begins with a quick introduction to wireless sensor nodes and Wireless Sensor Networks (WSNs). This is followed by a description of the protocol stack used in this work. The discussion then focuses on the advantage of introducing programming models, and their applications in WSN programming.

This chapter then goes on to present the concepts underlying Distributed Abstract Data Types (DADTs), a short description of existing DADT prototype [16], and an outline of the limitations of the current prototype. The next section presents Logical Neighborhoods (LN) [14], a mechanism that enables routing and scoping, and how LNs can be used to eliminate the limitations inherent in the DADT prototype. The chapter then concludes with a description of the simulation environment [5], [4] underlying the implementation, and the hardware platform - Sun Small Programable Object Technology (SPOT) [**?**]- used during the course of this work to experimentally validate the simulated implementation in a real-world environment.

2.1 Introduction to Wireless Sensor Networks

2.1.1 Sensor Nodes

Sensor nodes are multifunctional devices that are characterised by their low cost, low power consumption, and small form factor. They can communicate across short distances using Radio Frequency (RF) communication [2].[1]

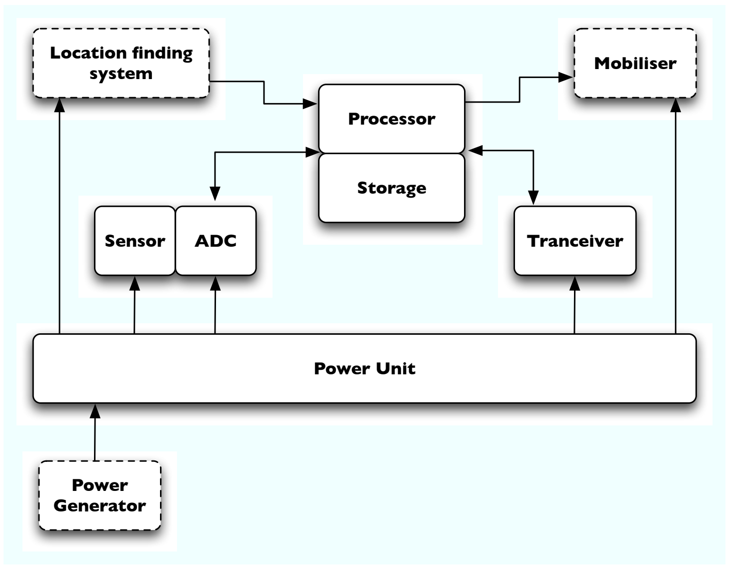


Figure 2.1: Architecture of a sensor node (reproduced from [2]). *Note: The presence of components drawn using dotted lines is application dependent.*

A typical sensor node is architected as follows [2]. It consists of a sensing unit, a processing unit, a transceiver unit, and a power unit as shown in Figure 2.1.1.

The sensing unit is comprised of sensors and analog-to-digital converters. The sensors may either be built into the nodes, or be hot pluggable [18]. The processing unit usually comprises a microcontroller/microprocessor that performs processing, and is associated with a storage unit. The transceiver unit facilitates

[1]Research is currently underway into investigating alternative techniques of wireless communication

node-node communication using a variety of techniques. The power unit provides the energy required to run the sensor node, and can use chemical batteries or power scavenging units such as solar cells. This can be seen in Figure 2.1.1.

Sensor nodes have constraints on both their size and their cost. The former constraint arises from the requirement that sensor nodes be easily deployable, while the latter arises from the requirement for fault tolerance (which in turn can be achieved only by being able to deploy cost-effectively large numbers of sensor nodes in the environment being monitored). These limit the memory capacity, processing power, and the amount of energy available on a particular node.

2.1.2 Wireless Sensor Networks

As mentioned in the previous section, sensor nodes are capable of communicating untethered with one another, and are hence capable of forming networks of nodes called a Wireless Sensor Network. WSNs are typically deployed randomly in an environment where phenomena are required to be monitored. A topology of a typical WSN has the following properties:

- A WSN is self-organising, given the random nature of the deployment.
- The WSN topology is subject to change. Sensor nodes should be capable of dealing with changes of this kind in order to deal 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 [2] is adapted from [3]. While ignoring the division of the stack into planes as irrelevant to the understanding of the work presented herein, one may view the protocol stack as consisting of the following layers (as can be seen from the Figure 2.2):

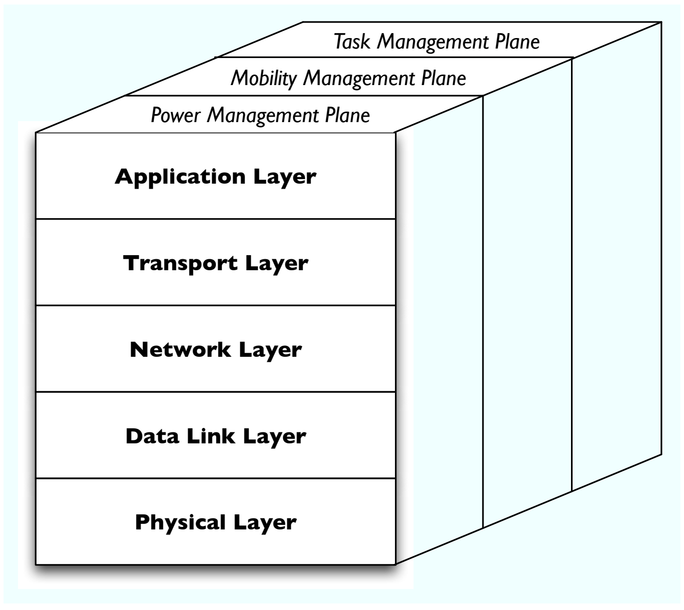


Figure 2.2: WSN protocol stack (reproduced from [2])

- *Physical Layer:* This layer is responsible for the transmission of data over the physical transmission medium.
- *Data Link Layer:* This layer deals with power-aware Medium Access Control (MAC) protocols that minimise collisions and transceiver on-time.
- *Network Layer:* This layer is primarily responsible for routing data across the network.
- *Transport Layer:* ??? - DOES LN USE IT?
- *Application Layer:* This layer holds the application software.

The LN mechanism that forms the bedrock of this work (see Section 2.6) provides the functionality of the Data Link Layer and the Network Layer (CHECK WITH LUCA). The concept of Distributed Abstract Data Types (see Section 2.4) is restricted to the application layer, and the focus of this work thus lies primarily within the application layer.

2.3 Programming models for WSNs

2.3.1 Motivation

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 primary problem with this 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 [15].

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 [15].

2.3.2 Benefits of using programming models

According to Yu et al [20], 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.

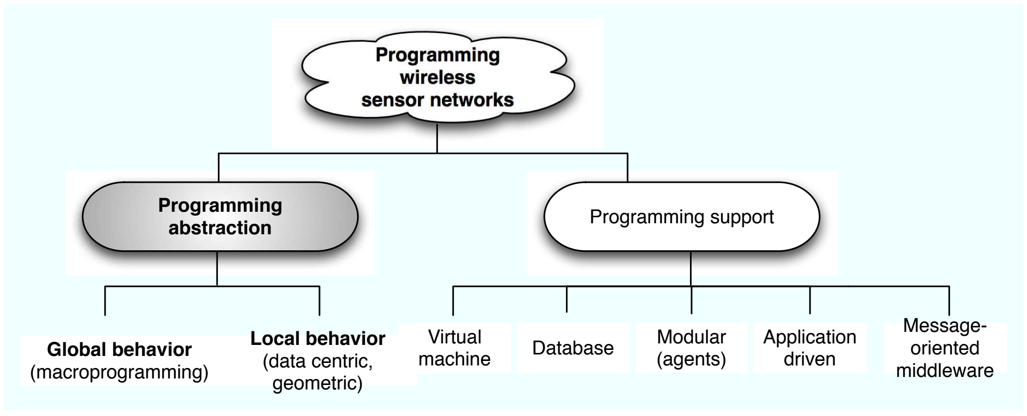

Programming wireless sensor networks
Programming abstraction
Programming support
Global behavior (macroprogramming)
Local behavior (data centric, geometric)
Virtual machine
Database
Modular (agents)
Application driven
Message-oriented middleware

Figure 2.3: Taxonomy of WSN programming models (reproduced from [9])

2.3.3 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 [9] (see Figure 2.3.3):

- *Programming support*, wherein services and mechanisms allowing for reliable code distribution, safe code execution, etc. are provided. Some examples of programming models that take this approach include Mate [11], Cougar [6], SOS [10], Agilla [7].

- *Programming abstractions*, where models deal with the global view of the WSN application as a system, and represent it through the concepts and abstractions of sensor nodes and sensor data. Some examples of programming models that take this approach include Kairos [8] and EnviroTrack [1] .

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.

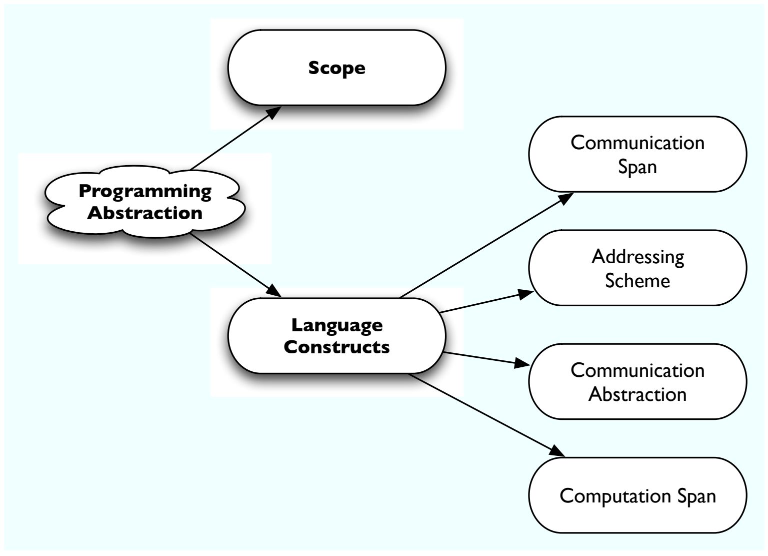


Figure 2.4: Classification of Programming Abstractions

2.3.4 Classification of WSN Programming Abstractions

Programming abstractions may either be global (also referred to as macroprogramming) or local [9].

In the former case, the sensor network is programmed as a whole, and gets rid of the notion of individual nodes [15]. Examples of macroprogramming solutions include *TinyDB* [12] and *Kairos* [8][2].

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

Programming abstractions may also be classified on the basis of the nature of the language constructs made available to the WSN programmer [15]. Some of the metrics used for classification are[3]:

[2]N.B.: Kairos does not do away with the notion of individual nodes as dedicated prorgramming constructs exist to iterate through the neighbours of a given node

[3]Only relevant to this work classification bases, from among those outlined in [15],were selected for discussion

- 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 [15] unless explicitly mentioned otherwise.

2.3.4.1 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:* Abstractions using this approach provide the programmer with constructs to allow nodes to exchange data with others within direct communication range.
- *Multi-hop group:* Abstractions that use this 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*.
- *System-wide:* When using abstractions of this kind, the programmer can 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 described above.

2.3.4.2 Addressing scheme

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

- *Physical addressing:* Nodes are identified using statically assigned[4] identifiers. The same address always identifies the same node (or nodes, if duplicate identifiers exist) at any time during the execution of the application.
- *Logical addressing:* Nodes are identified on the basis of predicates specified by the application programmer. Therefore, the same address (i.e., set of predicates) can identify different node(s) at different times.

2.3.4.3 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.

2.3.4.4 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:* 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.

[4]S: not entirely convinced. Unique is better, methinks.

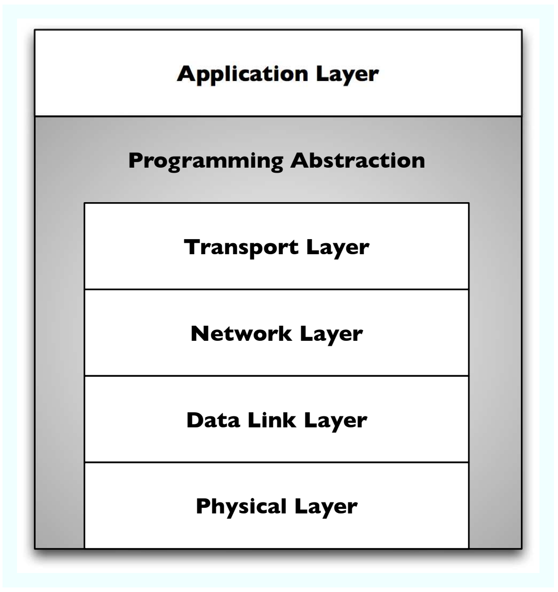


Figure 2.5: Programming models on the WSN protocol stack (adapted from [15])

- *Global:* An extreme case of previous type, a single instruction can impact every node in the WSN.

To illustrate, if it were possible in a given WSN programming abstraction to send a *Reset* message to all nodes with sensor readings greater than a specific threshold, the computation span enabled is of type *Group* (or possibly *Global*, which, as mentioned earlier, is an extreme case of the *Group* type).

2.3.5 Programming models on the WSN protocol stack

WSN programming models are placed between the application layer and the transport layer (change to network layer if trans layer not used) in the protocol stack shown in Section 2.2. As can be seen from Figure 2.3.5, fine-grained details are hidden from the application programmer's view. The abstracted details include:

- Higher-layer services such as routing, localisation, and data storage mechanisms (and optimisations).

- Lower-layers such as the MAC protocol used, and the physical means of communication such as RF communication.

2.4 Distributed Abstract Data Types

Distributed Abstract Data Types is a new programming language construct used to support distributed and context-aware applications. [16] introduces the concept of DADTs, and presents a prototype that implements it. The rest of this section discusses the concepts and the background, and provides the reader with an understanding of the prototype.

N.B.: Every code snippet provided as an example is a code snippet based on the DADT prototype described in [16].

2.4.1 Abstract Data Types

An Abstract Data Type (ADT) is the representation of a model that presents an abstract view to the problem at hand. The model of a problem defines:

- The data affected.
- The operations identified.

This set of the data values and associated operations, independent of any specific implementation, is called an ADT [**?**].

One of the simplest example of an abstract data type is a stack, which consists of the stacked data, and set of defined operations including *push(Data), pop(),* and *top()*.

As is clear from the example, several different implementations of an ADT may be defined from the proposed specification.

2.4.1.1 ADTs in WSNs

A WSN, as defined earlier, consists of several 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. The code snippet below shows the specification of a sensor ADT 2.1.

```
class Sensor {
  //data properties of the sensor
    int sensorType;
    double[] sensorReading;
  //operations that can be performed on the sensor
        public Sensor(type){
          type = sensorType;
          if (type == TEMPERATURE){
                sensorReading = new double[2];
          }     else if (type == HUMIDITY){
                sensorReading = new double[1];
          }
        }
    public double read(){ //read the sensor value(s).
          ...
        }
        public void reset(){ //reset the sensor
          ...
        }
}
```

Listing 2.1: Sensor ADT specification (reproduced from [16])

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

- An integer value to define the sensor type.
- An array of double values that holds the sensor reading. Depending on the type of the sensor, the number of values read varies.

and the following operations:

- A *read* operation, that reads the sensor readings.
- A *reset* operation, that resets it.

It is possible to define multiple such instances of this specification (for example, the specification may be defined as a Java interface that has to be implemented in order to create an ADT instance).

The ADT instances for a given sensor node that has two kinds of sensors - (a) a temperature sensor, that senses the atmospheric and in-device temperature readings, and (b) A humidity sensor, that senses a single value - may be defined using the ADT specification described above as shown in Listing **??**. This is quite similar to the declaration of the object of a class.

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

Listing 2.2: Sensor ADT instances

Thus, multiple ADTs can be used to abstract the nature of the sensor node, as well as the sensors therein, from the programmer.

2.4.1.2 Data and space ADTs

As a WSN consists of a collection of sensor nodes that are distributed in space, ADTs can also be used to describe the spatial location of a given sensor node. Thus, ADTs used in WSNs are of two types:

- *Data ADTs* are “conventional” ADTs which encode application logic (for example, 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” [16]. The space ADT may use different notions of space, such as physical location or netwo rk topology, depending on application requirements as determined by the programmer. A very limited notion of space is built into the DADT prototype in [16], by means of a superclass called *Site* which every Space ADT specification inherits from.

The ADTs presented in the previous section are examples of data ADTs. Space ADTs are defined and implemented in a similar manner, except for changes in their properties and operations (see Listing **??** for the specification of a Space ADT).

```
class SensorLoc extends Site {
  //space properties of the sensor
        location l;
  //operations that can be performed on the sensor
        public SensorLoc(Location l){
          this.l = l;
        }
        public double[] getLocation(){ //read the sensor value(s).
          ...
        }
        public void getBatteryLevel(){ //reset the sensor
          ...
        }
}
```

Listing 2.3: Sensor Space ADT specification (reproduced from [16])

2.4.1.3 Placement

As described in Section 2.4.1.2, a space ADT defines the computing environment that “hosts” the data ADT. A data ADT can be associated with the corresponding space ADT using a placement operation [5] as can be seen in Figure 2.4.2.4. A Sensor data ADT instance can be “placed” in a SensorLoc space ADT instance as shown in Listing 2.4.

```
  tempSensor = new Sensor(TEMPERATURE);
  sensorLoc = new SensorLoc(sensorGPSLocation);

  place(tempSensor, sensorLoc)
```

Listing 2.4: ADT placement

[5]the placement operation is performed explicitly by the application programmer.

2.4.2 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 ADTs in a distributed system are made collectively available using the interface of a DADT [16].

Using properties that are applied over multiple ADT instances, DADTs can be used to define views over the distributed state. Views allow for the dynamic restriction of the scope of distributed operations that the application requires to perform.

They 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:

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

The rest of this section presents the use of DADTs, specifically in relation to the DADT prototype [16].

2.4.2.1 DADT specification and instantiation

DADT specifications can be best understood by carrying forward the example described in Section 2.4.1.1. To allow for collective access to multiple ADT instances of the type specified in Listing 2.1, a DADT *DSensor* may be defined as shown in Listing 2.5.

```
datatype DSensor distributes Sensor with {
  operations:
        void resetAll();
        double average();
}
```

Listing 2.5: Data DADT specification (reproduced from [16])

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

- *resetAll():* Resets every sensor in the DADT view.
- *average():* Calculate the average of readings of every sensor in the view.

DADT specifications can be instantiated as an object of a class would, and can be used to perform distributed operations (see Listing 2.6).

```
DSensor ds = new DSensor();
double v = ds.average();
```

Listing 2.6: DADT Instantiation (reproduced from [16])

As can be seen from this listing, the distributed nature of the DADT can be abstracted from the application programmer.

2.4.2.2 Binding

The set of ADTs that are available for collective access using a DADT (in this case, the collection of ADTs of type Sensor) is called the *member set* of the DADT.

An ADT instance is made part of the member set by binding it to the DADT type. This is done using a dedicated programming construct as shown in Listing 2.7, where the Sensor ADT (see Listing 2.1) is bound to the DADT type *DSensor* defined in Listing 2.6.

```
bind(new Sensor(TEMPERATURE), ‘‘DSensor");
...
bind(new␣Sensor(HUMIDITY),␣‘‘DSensor");
```

Listing 2.7: Binding ADT instances to a DADT instance

2.4.2.3 Operators and Actions

Operators are used to allow distributed access and operations to be performed on ADT instances bound to a given DADT type. Operators are executed on the DADT instance, and may be of the following types:

- *Selection Operators:* Selection operators allow for an operation to be performed on a subset of the instances in the member set.
- *Conditional Operators:* Conditional operators make the code in the DADT method dependent on a global condition on the member set.
- *Iteration Operators:* These operators allow iterating over the ADT instances in the target set, and thus permit access to individual ADT instances.

Actions are constructs defined in the DADT type that causes the execution of an operation on a remote ADT instance. This can be understood by looking at the following example.

To perform a DADT read operation, an action is defined as shown in Listing 2.8.

```
public class DSensor_read_Action implements DADT.Action{

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

Listing 2.8: Defining a DADT action

This performs the operation on the ADT Instance of type *Sensor* (see Listing 2.1).

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 2.6), actions of the sort defined above are called[6]

2.4.2.4 Views

The concept of views is summarised in Figure **??**.

[6]The details of the implementation of actions are outlined in Chapter **??**.

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 [16]. Properties work on a principle similar to that of actions (see Section 2.4.2.3).

The member set may be partitioned into *DADT views* using properties. A DADT view may either be a (1) space view or a (2) data view. In the rest of this section, we focus on data views as space views are irrelevant to the understanding of this work.

To continue on the example running throughout this section, If the application programmer wished to partition the set of sensors bound to the DADT type *DSensor* in Listing 2.7 to refer to only those ADT instances that are temperature sensors, a Data View is defined as shown in Listing 2.9

```
DataView dv = new DataView(new DSensor_typeOf_Property(TEMPERATURE));

double[] averageTemperatures = ds.average(dv);
```

Listing 2.9: Definition and use of DADT Data View

The data view *dv* specifies that the scope of the DADT operation *average* is restricted to temperature sensors.

2.4.3 DADT prototype limitations

The prototype presented in [16] was developed as a proof of concept. The prototype uses IP multicast to simulate the lower layers of the protocol stack, thereby preventing its use in simulators or real nodes. This work sets out to correct this limitation by combining the DADT prototype with an implementation of logical neighborhoods.

2.5 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 discus-

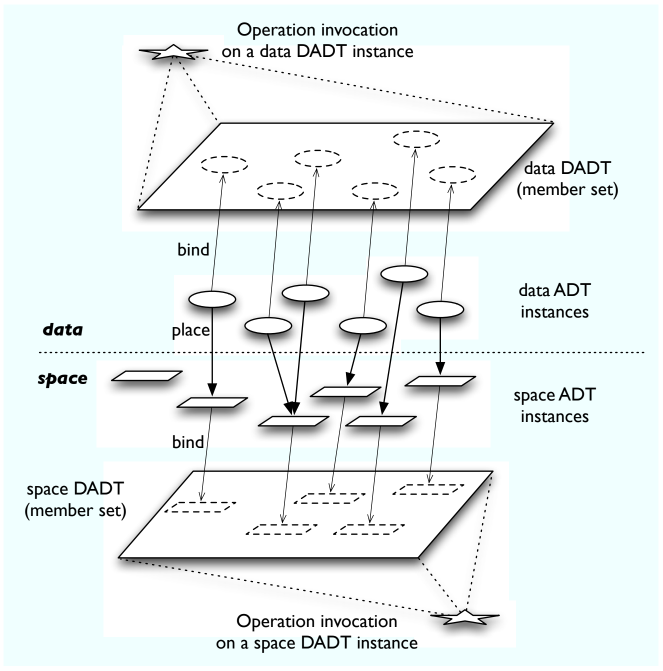


Figure 2.6: Data and space in the DADT model (reproduced from [16])

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

2.5.1 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 [17]:

- *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.

2.5.2 Java In Simulation Time (JiST)

JiST [5] 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[7], as can be seen in Figure 2.5.2.

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

```
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 2.10: Example JiST program (reproduced from [5]

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 2.11: Execution of the program in the JiST

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

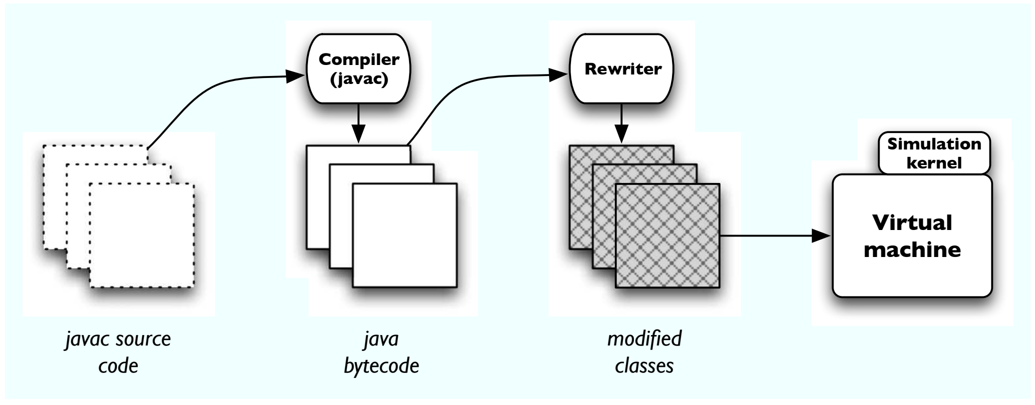


Figure 2.7: The JiST system architecture (reproduced from [5])

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.

2.5.3 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 [4], and is built upon the JiST discrete event simulator described in Section 2.5.2. It is organised a a collection of independent, relatively simple, event driven components that are encapsulated as JiST entities.

SWANS has the following capabilities [4]:

- 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 2.5.3.

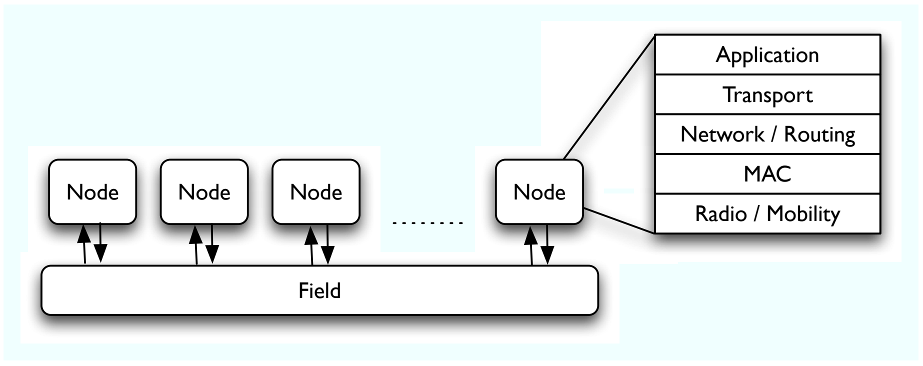
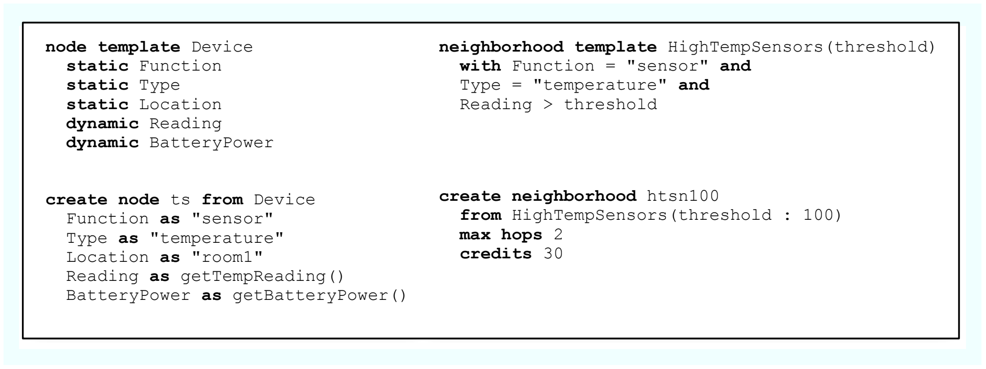


Figure 2.8: SWANS architecture

2.6 Logical neighborhoods

Please do not read. This section is being reworked, and is to be extended and written more clearly. The notion of neighborhoods gains importance with the increasing decentralisation of WSN applications, for example with the development of Wireless Sensor and Actor Networks (WSANs). In a decentralised application, the programmer is concerned with not just the implementation of the application itself, but also about identifying and interacting with the relevant subset of the network [13]. The latter task is far more complex, would require greater programming effort, and possibly less reliable code.

Mottola amd Picco propose the use of logical neighborhoods to allow for the partitioning of the network in applications of the type described above. A Logical neighborhood is a programming abstraction that differs from a physical neighborhood in that the notion of proximity is determined by application-dependent information and not by the communication range of the device.



```
node template Device
  static Function
  static Type
  static Location
  dynamic Reading
  dynamic BatteryPower


create node ts from Device
  Function as "sensor"
  Type as "temperature"
  Location as "room1"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

```
neighborhood template HighTempSensors(threshold)
  with Function = "sensor" and
  Type = "temperature" and
  Reading > threshold


create neighborhood htsn100
  from HighTempSensors(threshold : 100)
  max hops 2
  credits 30
```

Figure 2.9: Sample node and neighborhood definition and instantiation (reproduced from [13])

The system described in [13] provides an API that restricts communication to an application defined subset of the network, and a routing mechanism that performs better than existing ones in decentralised application domains. Logical neighbourhoods may be defined using a declarative language named SPIDEY.

An application may define a logical neighborhood as a collection of nodes that satisfy defined predicates. A node itself is described using a combination of static attributes (for examples, the type of the node) and dynamic attributes (for examples,the value of a sensor on the node).

Figure 2.6 shows examples of:

- A *node template* that specifies the attributes that define the node.
- A *node instance* that is created from a node template
- A *neighborhood template* that defines the predicates used to determine logical neighborhood membership.
- A *neighbourhood instance* that is created from a neighbourhood template, and provides specific values for the aforementioned predicates

The difference between the logical and physical neighbourhood of a given node is represented in Figure 2.6.

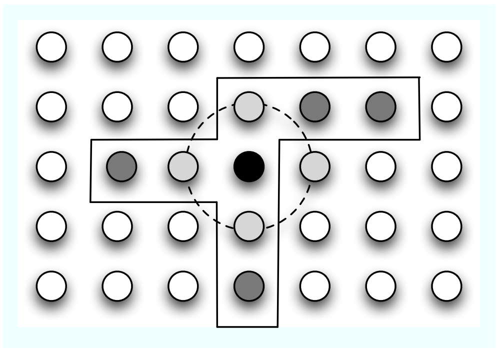


Figure 2.10: Representation of logical and physical neighborhood of a given node. *The dashed circle represents the physical neighborhood, whereas the solid polygon represents (one of) the node's logical neighborhood* (reproduced from [13])

2.7 SUNSPOTs

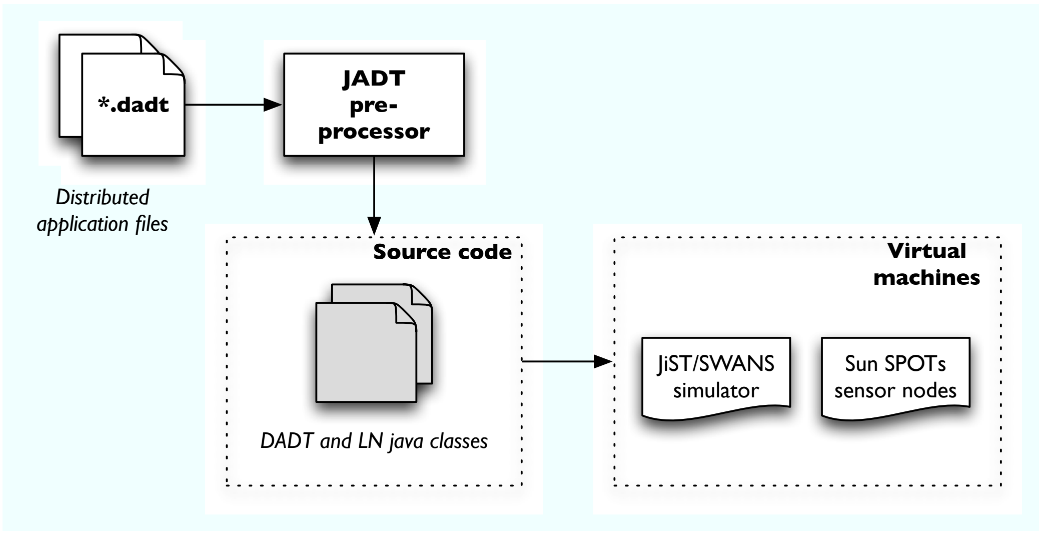


Figure 2.11: DADTLN prototype architecture - to be moved to the other section

Bibliography

[1] 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).

[2] AKYILDIZ, I., W.SU, Y. SANKARASUBRAMANIAN, AND E. CAYIRCI. A Survey on Sensor Networks. *IEEE Communications Magazine* (August 2002), 102–114.

[3] ANDREW S. TANNENBAUM. *Computer Networks*. Prentice Hall PTR, 2002.

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

[5] 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).

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

[7] 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.

[8] Gummadi, R., Gnawali, O., and Govindan, R. Macro-programming wireless sensor networks using Kairos. *Intl. Conf. Distributed Computing in Sensor Systems (DCOSS)* (2005).

[9] Hadim, S., and Mohamed, N. Middleware: middleware challenges and approaches for wireless sensor networks. *Distributed Systems Online, IEEE 7*, 3 (2006).

[10] 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).

[11] Levis, P., and Culler, D. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev. 36*, 5 (2002), 85–95.

[12] 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).

[13] 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.

[14] 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.

[15] Mottola, L., and Picco, G. P. Programming wireless sensor networks: Fundamental concepts and state-of-the-art. University of Trento, Italy.

[16] 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.

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

[18] WARRIER, S. Characterisation and Applications of MANET Routing Algorithms in wireless sensor networks. Master's thesis, School of Informatics, University of Edinburgh, August 2007.

[19] WEISER, M. The computer for the 21st century. *Scientific American 265*, 3 (1991).

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