A Gateway-Based Approach for Information Retrieval from Data-Centric Wireless Sensor Networks from IP Hosts

by

Brandon Keith Maharrey

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 13, 2010

Keywords: wireless sensor network, sensor network query, data collection, Directed Diffusion, data-centric

Copyright 2010 by Brandon Keith Maharrey

Approved by

Alvin S. Lim, Chair, Associate Professor of Computer Science & Software Engineering
John A. Hamilton, Jr., Professor of Computer Science & Software Engineering
Xiao Qin, Associate Professor of Computer Science & Software Engineering

Abstract

With applications ranging from environmental and health monitoring to military surveillance and inventory tracking, wireless sensor networks (WSNs) are changing the way we collect and use data and will be a major part of our technological future. The decreased manufacturing cost of these small devices has made it reasonable to deploy many sensor nodes — 10s to 1000s and more — over large and small indoor and outdoor areas for sensing tasks. With this increase in density come data-gathering problems. It would be useful if an IP-based host could collect information from multiple remote data-centric networks using a predefined application programming interface (API). This thesis presents two APIs, one API for wireless sensor nodes in a data-centric Directed Diffusion WSN and the other API for IP-based nodes residing on an IP network outside of the WSN. An associated WSN middleware layer used to effectively connect these two APIs is also presented. These three components work together in harmony to enable IP-based hosts to gather sensed data from one or more remote WSNs through an application-layer gateway which links these different remote WSNs to an IP network.

Acknowledgments

I would like to thank Drs. Alvin S. Lim, John A. Hamilton, Jr., and Xiao Qin for their guidance in the writing of this thesis. I would also like to thank Raghu K. Neelisetti, Qing Yang and Arunkumar Thippur Jaykeerthy for their efforts in acclimating me to the life of a graduate student and helping me when the times got tough. It has certainly been a fun ride.

I also cannot forget my wife, Tasha M. Crawford, my step-daughter, Rosemary M. Crawford-Goosby, my mother, Carol Elmore and my sister, Brandi N. Maharrey, who have all been there as my support system throughout graduate school and this research.

Thank you all very much.

Table of Contents

Abstract ii

Acknowledgments iii

List of Figures vii

List of Tables x

List of Abbreviations xi

1 Introduction 1

2 Problem Statement 4

3 Objective 6

4 Motivation 8

5 Challenges 10

6 Limitations 12

7 Related Work 14

7.1 Gateway- or Proxy-Based Approaches 14

7.2 IP-Enabled Approaches 16

7.3 Overlay Approaches 18

7.3.1 Sensor Network Overlay IP Network 18

7.3.2 IP Network Overlay Sensor Network 19

7.4 6LoWPAN and IEEE 802.15.4 Standards 20

7.5 Data Collection from WSNs 21

7.5.1 Clustering, Routing and Data-Centric Storage Protocols 22

7.5.2 Query Optimization 24

7.5.3 Middleware Solutions 24

8 System Overview 26

8.1 Directed Diffusion . . . 29
8.1.1 Directed Diffusion Basics . . . 32
8.1.2 Directed Diffusion and the Publish/Subscribe Messaging Paradigm . 35
8.1.3 Directed Diffusion Sensor Node Application Programming . . . 35
8.2 Dynamic Services . . . 36
8.3 Device Roles . . . 38
8.3.1 External Agents . . . 38
8.3.2 Sensor Nodes . . . 38
8.3.3 Gateway Node . . . 40
9 Dynamic Services . . . 41
9.1 Dynamic Services Include Structure . . . 41
9.2 Inside Dynamic Services . . . 44
9.3 Dynamic Services API Available to Sensor Role . . . 47
9.4 DS API Example Usage . . . 51
9.4.1 Simple Producer API Usage Details & Example . . . 51
9.4.2 Complex Producer API Usage Details & Example . . . 55
10 External Agent Role . . . 63
10.1 External Agent API . . . 63
10.2 External Agent API Example Usage . . . 65
11 Gateway Role . . . 68
12 Integrating the WSN with the IP Network . . . 70
12.1 Building a Sensor Network Application . . . 70
12.2 Building an External Agent Application & Performing a Query . . . 74
13 PC104 Testbed & Case Studies . . . 77
13.1 PC104 Testbed Specifications . . . 77
13.1.1 Hardware . . . 77
13.1.2 Operating System . . . 82

13.2 Sample Case Studies . . . 82
13.2.1 Sensor Node Applications . . . 88
13.2.2 External Agent Applications . . . 94
13.3 Target Tracking Experimental Results . . . 98
14 Experiment Design & Performance Evaluation . . . 102
14.1 Experiment Design . . . 102
14.2 Performance Evaluation . . . 103
14.2.1 LOC Metric . . . 103
14.2.2 DS Delay Analysis . . . 104
15 Future Work . . . 111
15.1 Concerning Sensor Applications . . . 111
15.2 Concerning the Application-Layer Gateway . . . 112
15.3 Concerning External Agent Applications . . . 112
16 Conclusions . . . 113
Bibliography . . . 115

List of Figures

1.1 "Tell" Gateway 2

3.1 "Tell" Gateway 6

7.1 Gateway- or Proxy-Based Approach 15

7.2 TCP/IP-enabled Approach 17

7.3 Packet Translation at Overlay Gateway Node in Sensor Overlay IP Network . . 19

8.1 Network Architecture Showing Heterogeneous IP Hosts 27

8.2 Network Architecture Showing IP Hosts and Multiple Wireless Sensor Networks 28

8.3 An External Agent Queries a WSN 29

8.4 An External Agent Receives Data from a WSN 30

8.5 Network Stack of Device Roles 31

8.6 Stages of Interest Propagation in Directed Diffusion 33

8.7 Stages of Data Propagation in Directed Diffusion 34

8.8 Pure DD Main & Tasking Threads Pseudo-Code 36

8.9 Comparison of Networking Stacks 37

8.10 External Agent Stack with Many Applications 39

8.11 Sensor Node Stack with Many Applications 39

8.12 Gateway Node Networking Stack 40

9.1 Dynamic Services, Gateway Node, Sensor and External Agent Applications Include Dependency Graph 42

9.2 Dynamic Services Message Queue Illustration 46

9.3 Dynamic Services Main & Tasking Threads Pseudo-Code 46

9.4 Simple Producer Tasking and Data Production 52

9.5 Simple Producer Detail Message Passing 53

9.6 Complex Producer Tasking and Data Production 57

9.7 Complex Producer Detail Message Passing 58

13.1 PC104 CPU Module 78

13.2 PC104 PCMCIA Module 78

13.3 PC104 Power Supply Module 80

13.4 Orinoco Gold Wireless PCMCIA LAN Card 80

13.5 Lucent Omnidirectional External Antenna 81

13.6 Compact Flash Card 81

13.7 PC104 RAM 81

13.8 PC104 USB Microphone 83

13.9 PC104 +12V AC Power Supply 83

13.10 PC104 +12V Battery 84

13.11 Completed PC104 Assembly without Casing 84

13.12 Completed PC104 Assembly with Casing 85

13.13 Close-Up of Completed PC104 Assembly with Casing 85

13.14 WSN Example Deployment Scenario 87

13.15 Sony EVI-D30 Pan-Tilt-Zoom Camera 88

13.16 WSN Experiment Setup 89

13.17 Sound Analysis Image 91

13.18 Target Tracking Results 100

13.19 Snapshot of Moving Target 101

14.1 Tasking Time Explanation with Dynamic Services 105

14.2 Publishing Time Explanation with Dynamic Services 106

14.3 Tasking Time Explanation without Dynamic Services 107

14.4 Publishing Time Explanation without Dynamic Services 108

14.5 Dynamic Services Tasking Time Performance Graph 109

14.6 Dynamic Services Publishing Time Performance Graph 110

List of Tables

9.1 Definition of Tasking Thread and Main Thread using Pure DD vs using DS . . 45

13.1 Target Tracking Experimental Results 99

14.1 Lines-Of-Code Comparison With & Without Dynamic Services 104

List of Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
ACM	Association for Computing Machinery
API	Application Programming Interface
CPA	Closest Point of Approach
CPU	Central Processing Unit
CTP	Collection Tree Protocol
DARPA	Defense Advanced Research Projects Agency
DCS	Data-centric Storage
DD	Directed Diffusion
DS	Dynamic Services
EA	External Agent
GPS	Global Positioning System
GPSR	Greedy Perimeter Stateless Routing
GSN	Global Sensor Network
GUI	Graphical User Interface
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IP Internet Protocol

IPv6 Internet Protocol version 6

LAN Local Area Network

LEACH Low-Energy Adaptive Clustering Hierarchy

LOC Lines of Code

LR-WPAN Low-Rate Wireless Personal Area Network

MPI Message Passing Interface

OS Operating System

PC Personal Computer

PCMCIA Personal Computer Memory Card International Association

PDA Personal Data Assistant

PEDQPP Power Efficient Data Query Processing Protocol

RFC Request For Comment

RMST Reliable Multi-Segment Transport

SBC Single-Board Computer

SCADDS Scalable Coordination Architecture for Deeply Distributed Systems

SPIN Sensor Protocols for Information via Negotiation

SQL Structured Query Language

STL Standard Template Library

TAG	Tiny AGgregation
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Chapter 1

Introduction

A wireless sensor network mainly consists of many independent, low-power, low-cost devices capable of sensing, processing and wireless communication [7]. Their main purpose is to collect and disseminate environmental data and possibly perform some calculations [4]. There has been a push, especially in industry[1], in recent years to make real-time data collected from WSNs more readily available to consumers of this information. However, there are no convenient tools or specific frameworks in place to allow instant access to this sensed information in a programming environment. Thus, one of the main problems with deploying WSNs is gathering the data they produce and using it in flexible ways.

Assume, for the moment, you are an application programmer and your job is to, some how, gather data from hundreds of nodes located in several WSNs located around the world and use this data in some application. You already know it is going to be a tough job, considering the WSN nodes are spread around the world. You realize this is going to be a job best suited for IP communication since all of the remote locations are already connected to the Internet. Why not use the already-in-place communication infrastructure provided by the Internet, right? It then dawns on you that the overhead incurred by TCP or UDP and IP may be prohibitive for use on your resource-constrained sensor nodes in the remote WSNs. However, that is not a real issue since you are aware of the existence of data-centric networking protocols. Data-centric networking protocols can significantly reduce wireless communication overhead in WSNs because there is no wasted overhead in transmitting end-point identification information along each wireless hop [5]. You are even more sure of your decision to use a data-centric networking protocol when you notice all of your data can be

[1]See http://www.archrock.com/company

easily named and divided into categories, e.g., humidity, pressure, temperature, vibration, light intensity, etc.

Would it not be amazing if you, as the IP-based application programmer, could simply "tell" a remote WSN which type or category of data you wanted and have it delivered to your application by the IP network? That is precisely the solution this thesis attempts to provide — a simple way for a programmer of IP-based applications to simply "tell" a data-centric WSN which data he wants and have it delivered to his application via the IP network. This concept is illustrated in Figure 1.1.

It is, however, not quite as simple as this first glance. There must be some sort of interface between the Internet host and the WSN to facilitate this communication.

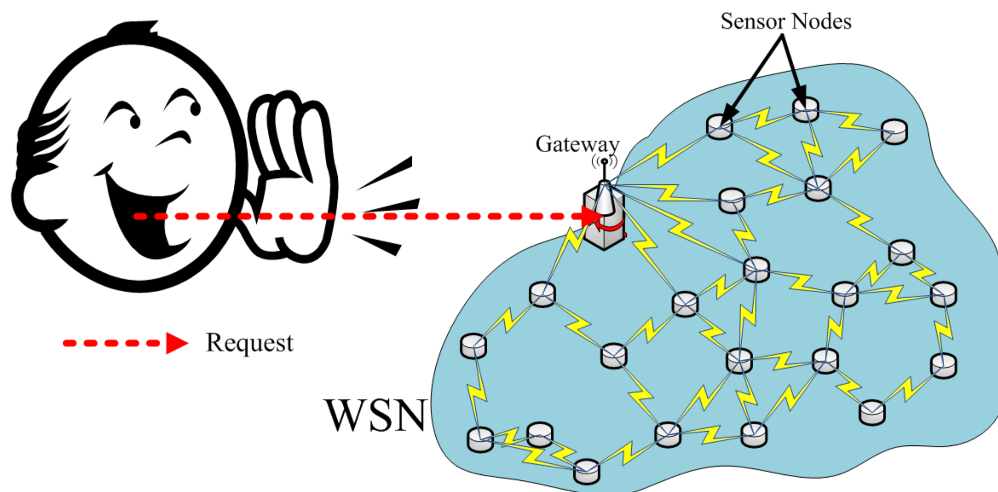


Figure 1.1: This figure shows the optimal process where an application simply requests — by name — named data from a remote wireless sensor network.

The proposed middleware layer and gateway node employed as the access point for the purpose of information retrieval from WSNs work together to ease the implementation of sensor network applications in the WSN by providing a standard interface to the data-centric WSN networking protocol and IP-based applications alike. Further, the approach described in this thesis does not require any changes to any existing data-centric WSN protocols nor the IP protocol. The middleware layer for the WSN could easily be adapted to other data-centric

WSN protocols, and the IP-based application programmers' API could not only be used for computers, laptops and servers but also other smaller devices such as PDAs, SmartPhones and other IP-based devices.

In the following chapters of this thesis, a concise problem statement is defined in Chapter 2 and associated objective given in Chapter 3. Motivation is outlaid in Chapter 4, and challenges and limitations of this research regarding the gathering of data from one or more WSNs from IP-based hosts are presented in Chapters 5 and 6, respectively. Following these, a discussion of other relevant approaches to gathering data from WSNs connected to the Internet is presented in Chapter 7. An overall summary of the work is then given, including discussing Directed Diffusion and Dynamic Services and defining roles of various agents within the system in Chapter 8. Following this summary, details of Dynamic Services and its relation to the sensor, External Agent and gateway roles is given in Chapters 9, 10 and 11, respectively. A description is then given of how all of the actors work together within the system in Chapter 12. Next, details of a PC104 testbed and WSN sensor node case study application is given along with two IP-based applications in Chapter 13. Then, experimental design details are given along with a few metrics of interest in Chapter 14 to show that the system is feasible. Finally, a few avenues of future research and final concluding remarks are given in Chapters 15 and 16, respectively.

Chapter 2

Problem Statement

When an application programmer is tasked with gathering data from possibly thousands of remote sensor nodes from one or multiple WSNs located locally or remotely, the problems are many. The decision to use the existing IP infrastructure may be obvious considering the ubiquitous access to the Internet, but there are more subtle problems like how to actually query remote sensor nodes for their data, not to mention the burden on the WSN and application programmers of programming in this environment without many tools at their disposal. This thesis attempts to provide solutions for these problems. For the task of actually querying the remote WSNs, a working API available to IP-based application programmers for querying one or more remote WSNs is given. A related API is also provided for WSN sensor node application programmers for programming on the sensor nodes in the WSN.

With these solutions come more problems. Through what means or mechanisms do the IP-based hosts actually retrieve the data from the remote WSNs? In this thesis, a discussion of the implementation of an application-layer gateway is given which simplifies the process of retrieving information from the remote WSNs. Because the WSNs discussed in this thesis are data-centric and hosts residing on a host-centric network (the IP network) wish to gather data from these WSNs, there is the problem of translating the information contained in this data-centric network to a network which is address-centric. Therefore, another problem this application-layer gateway must solve is how to translate named data coming from the data-centric WSN to an IP address so that transmission of this data can occur via this address-centric IP network and arrive at the correct IP-based host(s). Simply

explained, there should be some translation or mapping service provided by the gateway node to translate named data to an IP address.

One more problem is how IP-based application programmers actually query the remote WSN(s). How is data in the WSN requested by an IP-based host? Since the data-centric network discussed in this thesis is based on the publish-subscribe paradigm, IP-based hosts must take this fact into account. Subscriptions must be sent from IP-based hosts to the application-layer gateway node in some fashion. The gateway node must then distribute this subscription to other sensor nodes in the WSN for data production to begin at sensor nodes who "know" how to produce this named data.

Chapter 3

Objective

This thesis presents two APIs and an associated middleware layer which attempts to make both IP-based and sensor node application programming easier for the purpose of harvesting data from multiple remote data-centric WSNs. The two APIs work together to make data in remote data-centric WSNs more readily accessible. Please see Figure 3.1 for an illustration of how these APIs work together with the associated middleware layer to allow IP-based hosts to gather sensed data from sensor nodes.

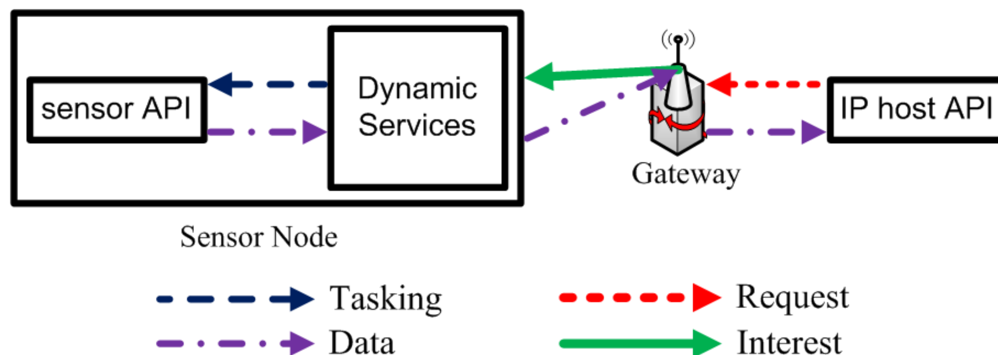


Figure 3.1: This figure shows the optimal process where an application simply requests — by name — named data from a remote wireless sensor network.

At this time, there is no standard method of or API for gathering data from a data-centric WSN and transporting that named information from the remote WSN to IP-based hosts who are interested in this data. Therefore, a main objective is to give sensor node and IP-based application programmers a tool — in the form of two APIs — to enable easy access to data sensed by WSN sensor nodes in a programming environment on IP-based hosts.

As previously mentioned, two simple APIs are presented — one for IP-based application programmers and the other for sensor node application programmers. The API for IP-based

application programmers is used to enable communication between IP-based hosts and the gateway node connecting the IP network to the WSN. This API provides two different services to IP-based hosts: a method to submit requests to and a related method of receiving information from the remote WSN(s).

The other API is used to facilitate communication between sensor node applications and Dynamic Services (DS), a middleware layer written on top of the data-centric networking protocol Directed Diffusion (DD). Without DS, tasking of sensor node applications by IP-based hosts could not occur nor could transmission of information take place from sensor node applications to the gateway node. This API also makes the programming job easier for sensor node application programmers and thereby makes adding sensing capabilities easier to implement.

A description is also given of an implementation of an application-layer gateway used as the interface to the WSN for the purpose of information retrieval from data-centric WSNs via IP-based hosts. Not only is this application-layer gateway node the access and data transfer points which enables access to the remote WSN(s), it also provides a *data name-to-IP address* mapping service. This service essentially uses an internal mapping structure to decide to which IP-based host(s) named data should be forwarded from the WSN.

Since there are many different device roles that must work seamlessly together to make a system like the one discussed in this thesis work well, there are a few networking architecture assumptions given and discussed.

Finally, results are given of a lines-of-code (LOC) metric and a timing analysis of DS. The LOC metric is used to determine the relative ease of the job of the sensor node application programmer when compared to programming a sensor node application directly over DD instead of using the services provided by DS. The timing analysis is used to show that the overhead incurred by DS and the sensor node API is negligible when compared to using the pure DD API.

Chapter 4

Motivation

The area of sensor networking applications is exploding rapidly. In the recent past, many new sensor networking applications have surfaced in the literature, and most notably among them are wildlife habitat monitoring [2], forest fire detection [30], alarm systems [1] and monitoring of volcanic eruptions [11]. These scenarios involve many unique issues and challenges still remain when faced with the problem of gathering this sensed data in real time for analysis, computation or storage. IP-based application programmers themselves are faced with the difficult problem of interconnecting sensor networks with IP-based hosts. Therefore, a main motivation of this research is to ease the data-gathering problems that IP-based programmers face when gathering data from one or more remote WSNs.

DS, a middleware solution built on top of the DD networking protocol, attempts to alleviate some of the programming and data-gathering problems associated with gathering data from a data-centric WSN from IP-based hosts by providing a neat and clean interface to the DD network. This DS service layer is also meant to conserve energy on the sensor nodes by eliminating the polling feature of pure DD sensor applications. Also, through employing a gateway-based approach for the task of interfacing with and information retrieval from the WSN, the solution provided in this thesis allows IP-based hosts to gather data from one or multiple remote WSNs concurrently with an easy-to-use API. Therefore, a major motivation for building a system of this type would be to decrease time-to-market of various applications which depend how much time application programmers spend on various programming tasks by simplifying said programming tasks.

A motivation for using an application-layer gateway can be found by analyzing more closely a statement found in [21]. In [21], Zhang *et al.* claim that an application-layer

gateway-based approach, the approach employed in this thesis, is application-specific. Let this statement stand disputed with this: a well-built application-layer gateway is not necessarily application-specific and can be used in a variety of different applications. The real power comes from the flexibility of the gateway node and the interface used to facilitate communication through that gateway node. In this thesis, the application-layer gateway presented is a protocol converter which is not application-specific. Therefore, any new application can be deployed over the WSN or developed on the IP network side with no reprogramming of the gateway node for each new sensor application deployed.

Finally, the literature discusses various SQL-like (Structured Query Language) WSN query languages for querying WSNs. The services provided by the application-layer gateway-based approach described in this thesis could be used as a back end for these SQL-like WSN query implementations. The SQL-like query implementation could use the API provided in this thesis to actually perform the querying of one or more remote WSNs. As an example scenario, an Internet user could submit SQL-like queries to his local machine located on the IP network. This Internet-based machine would have the task of translating that query into a named interest destined for one or more WSNs. The machine would then send that interest to one or more application-layer gateways located around the Internet and possibly receive data from the WSN(s). The IP-based machine could then display the information received from the WSN(s) to the user according to the initial query.

Chapter 5

Challenges

There are a few challenges that programmers face when trying to determine how to harvest information from a remote WSN. This thesis partly focuses on the challenge of actually retrieving named data from possibly many remote data-centric WSNs and delivering that information, via IP networks, to interested IP-based hosts. Therefore, IP-based and sensor node programmers must in some way agree on the mapping from the "name" of the data and the structure which embodies the actual data. Put in the form of a concrete example, if an IP-based host wishes to query a sensor network for the named data "TEMPERATURE," the sensor node should recognize "TEMPERATURE" and produce the data which corresponds to the *data definition* or *data structure* of that named data. It is up to the sensor network programmer to ensure that data produced by sensor nodes in the WSN matches the data name requested by IP-based hosts — that is, that the IP-based host receives the correct and proper data he thinks he is receiving from the WSN. In other words, IP-based programmers and sensor node programmers must be in agreement regarding the actual data structure which is produced by sensor nodes in the WSN and the data structure which is consumed by IP-based hosts on the IP network. A mismatch in the way the sensor node programmer or the IP-based programmer interprets or defines the data definition or data structure can be a major pitfall and cause of application errors.

This thesis assumes a data-centric networking protocol in the WSN and consequently implies that the TCP/IP protocol stack is not available in the sensor nodes. Consequently, another issue that may arise due to the missing TCP/IP protocol stack is reliable delivery of large files. However, there are reliable transport protocols for data-centric networking protocols which can be used to accommodate reliable routing of large files inside the data-centric

network. Reliable Multi-Segment Transport (RMST) is an example of such a transport protocol for use in a Directed Diffusion-based WSN [9]. After RMST has routed and reassembled the data at the gateway node, the gateway node could then, by way of TCP and IP, reliably transmit the large file(s) to IP-based host(s).

Another challenge of this research concerns IP addressing of gateway nodes around the Internet, and specifically, how IP-based hosts determine the IP addresses of gateway nodes in order to query or task the WSN. This thesis leaves this an open problem, but does give a suggestion. Consider an application of a WSN in which weather information is gathered from several remote WSN locations by a weather broadcasting company. In this case, the company's administrative unit itself would be responsible for keeping track of the IP addresses of their various remote WSN gateway nodes and delivering this information to their programming team(s). An assumption of this thesis is that IP-based hosts *some how* have access to the IP address information of remote WSN gateway nodes, whether it be through some administrative unit of a company, like in this example, or some other yet-to-be-designed WSN gateway node discovery protocol.

Most challenging of this research was, by far, the creation of Dynamic Services. DS's role in the system is very important and should not be overlooked or taken lightly. Without DS, tasking of the sensor nodes would not be possible given only a "*data name*," which originates from the IP-based hosts. DS serves to, essentially, tie the IP-based hosts' API to the sensor node API and allow data to be transferred from each sensor node application to IP-based hosts via the application-layer gateway node. Again, see Figure 3.1 for a simplified illustration.

Chapter 6

Limitations

This thesis employs an application-layer, gateway-based approach for information retrieval from data-centric networks from IP-based hosts. While this technology will aid in simple and efficient information retrieval, it does have its limitations.

In this approach, the full TCP/IP networking stack is not available to sensor nodes, making remote reprogramming or administrative tasks on the individual sensor nodes very difficult or impossible. However, if the sensor nodes are pre-programmed with all of the necessary software, settings and parameters, this should not be an issue. It should be said that before any group deploys the actual WSN, all sensor nodes should be programmed beforehand with all necessary components already in place. This is actually not such a problem and makes logical sense. The reason is that since IP-based hosts must know the format of the data for which they are querying the WSN, it would be difficult and annoying to users of the IP-side API, especially if the system were widespread with many active IP-based hosts frequently querying the WSNs, to change or update all IP-based hosts' data type definitions each time a change is made to any data type(s) inside the WSN.

This thesis assumes that there is an operating system on the WSN nodes, and, thus, assumes that the sensors are at least SBCs capable of running at least a small version of the Linux operating system (OS). Therefore, this research is not specifically meant to be used in deeply embedded WSNs. This assumption puts a lower bound on the size and power of the sensor node because the smallest of current sensor nodes, the Mica2Dot and SmartDust nodes, for example, are not capable of running a complete OS due to energy, memory and processing constraints.

Considering mobile data-centric networks, although the data-centric networking protocol investigated in this thesis is not specifically designed for mobile WSNs, a similar implementation of DS over a mobile data-centric networking protocol could be realized.

This thesis does not provide nor does it claim to provide any WSN discovery protocol. Administrators of the remote WSNs themselves must keep track of the IP addresses and port numbers of the remote gateway nodes. On the same token, there is also no WSN *data* discovery protocol. With the current system, administrators also must keep track of the types of data that each remote WSN can produce. However, there is at least one simple, but not currently implemented, solution to the latter problem of WSN data type discovery. The application-layer gateway node could be given the responsibility of keeping track of which types of data are available to be produced in each remote WSN to which it is the gateway. When an IP-based host wishes to retrieve a certain type of data from a remote WSN, it could first query the remote WSN gateway node to determine if that remote WSN can, in fact, produce the given type of data. If the response returned from the gateway node answers in the affirmative, then the IP-based host could then submit a request for that data type. The gateway node could be made aware of these data types by the sensor nodes residing inside the WSN. When each sensor node application starts up, it could send a registration message to the gateway node signaling the type of data it intends to produce. The gateway node should maintain a list of all of these data type names which the local sensor node applications can produce. In this way, the IP-based hosts can determine which data types can be produced by remote WSNs before submitting requests. The gateway node could also send a list of all data types its associated WSN can produce and let the IP-based host decide which data type(s) it wishes to consume by looking into this list.

Chapter 7

Related Work

Many researchers have done previous work on interconnecting wireless sensor and IP networks and gathering data from WSNs. The majority of the techniques, like the one presented in this thesis, treat WSNs as a separate entity from the Internet [13]. The techniques are divided into two main approaches: a gateway-based approach and an approach in which all sensor nodes are TCP/IP-enabled — that is, capable of direct, end-to-end communication with IP-based hosts.

7.1 Gateway- or Proxy-Based Approaches

The most common approach to connecting a WSN with an IP network is through a gateway or proxy node. In this approach, the gateway node acts as a relay to translate and forward packets from one network to the other [3, 17, 25]. The authors of [3] describe two gateway-based approaches: using the gateway as a relay or as a front-end. When the gateway acts as a relay to the WSN, it simply relays any information from the WSN to any registered IP-based host that wants that information — the approach taken in this thesis. When the gateway node acts as a front-end to the WSN, it actively collects and stores data from the WSN in some kind of database that users can query with SQL-like query languages. A representative networking architecture of a general gateway-based approach is shown in Figure 7.1. In this approach, a gateway or proxy node is directly connected to both the IP and wireless sensor networks and acts by translating packets from the IP networking protocol to the WSN networking protocol and vice versa.

[25] notes a challenge in gateway-based approaches. Specifically, Song *et al.* in [25] suggest that the gateway node can be a bottleneck to the flow of network traffic, especially

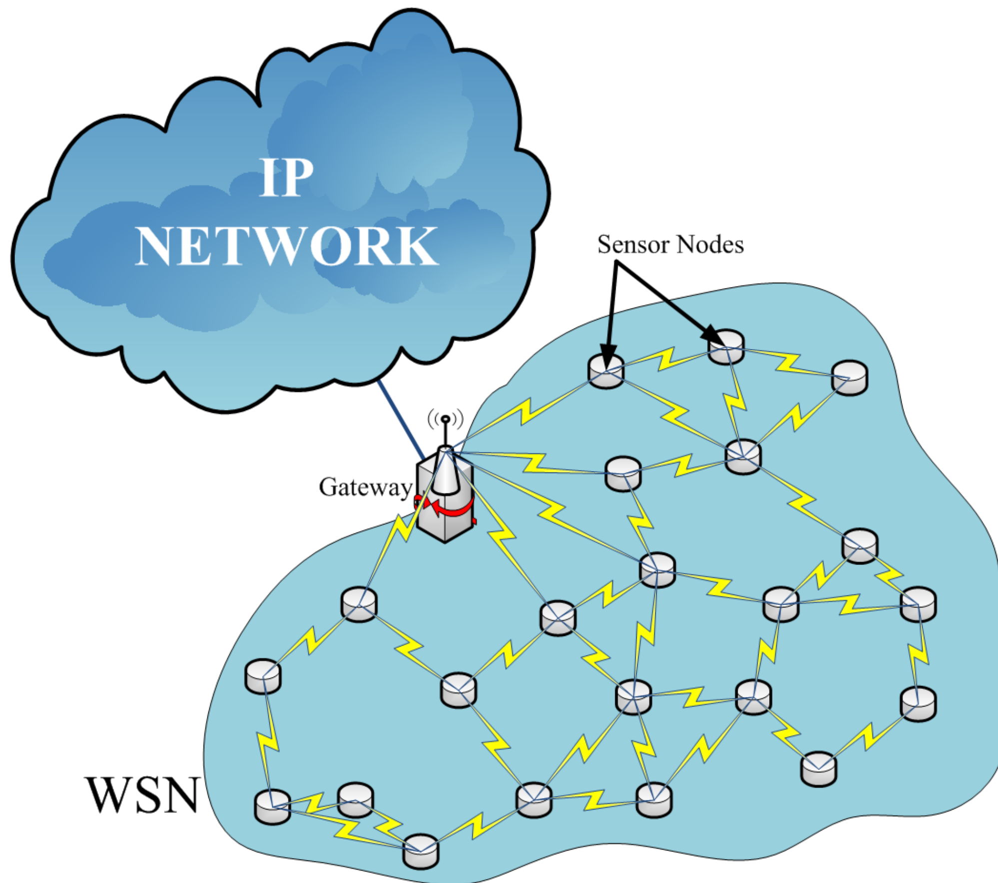


Figure 7.1: This figure represents the network architecture of an approach to connecting a WSN to the Internet via a gateway node. The gateway node communicates wirelessly with the nodes in the WSN within its communication range. The gateway node is responsible for translating packets from the IP networking protocol to the WSN networking protocol and vice versa.

if a surge of data need to be transmitted from the gateway node to an IP-based host, and consequently developed an application-layer gateway based on the PXA270 SBC using embedded Linux designed to meet this throughput requirement. The authors, however, do not mention any specific WSN routing protocol and largely focus on the hardware and software requirements of a gateway node specifically designed to transmit TCP/IP packets from the WSN to the IP network. This suggests WSN nodes should be TCP/IP enabled, but leaves the problem of actual gathering of the data from the sensor nodes an open issue.

One advantage of gateway-based approaches is the two communication networks are totally decoupled, allowing for specialized protocols to be implemented in the WSN [3]. The gateway node can also act as a mediator for WSN data transmission — and in the case of this thesis, named interest acceptance or denial by the gateway node — by implementing security features such as user and data authentication [3].

Both the relay and front-end gateway-based approaches, however, suffer from the fact that direct access to individual sensor nodes is impossible. This does not represent a problem as far as gathering data is concerned; however, if it is necessary to reprogram or administer sensor nodes, this may be an issue. Further, schemes generally based on this gateway- or proxy-based approach do not necessarily ease the task of gathering sensor node data alone.

7.2 IP-Enabled Approaches

Besides gateway-based approaches to interconnecting wireless sensor and IP networks, IP-enabled WSNs have also been explored. One approach of this type assumes a full TCP/IP stack on each sensor node. In this approach, the WSN is directly connected to the IP network to enable direct communication between WSN sensor nodes and IP-based hosts. A representative networking architecture of a general IP-enabled approach is shown in Figure 7.2.

The main advantage of using TCP/IP in this way is that now there is no need for protocol conversion or gateways. However, the overhead for the full networking stack on an energy-constrained sensing device may be prohibitive, especially when the end-to-end retransmissions incurred by the TCP protocol cause even more undue retransmissions at intermediate nodes. It has been shown that the majority of energy in a WSN is used for wireless communication [10]. Therefore, if one considers the protocol overhead for TCP/IP networks in the context of WSNs, it can be seen that this overhead is prohibitive. As an illustrative example, consider the fact that in IPv6, packet overhead is more than 40 bytes per packet. What impact does this have on the cost of radio transmission compared to

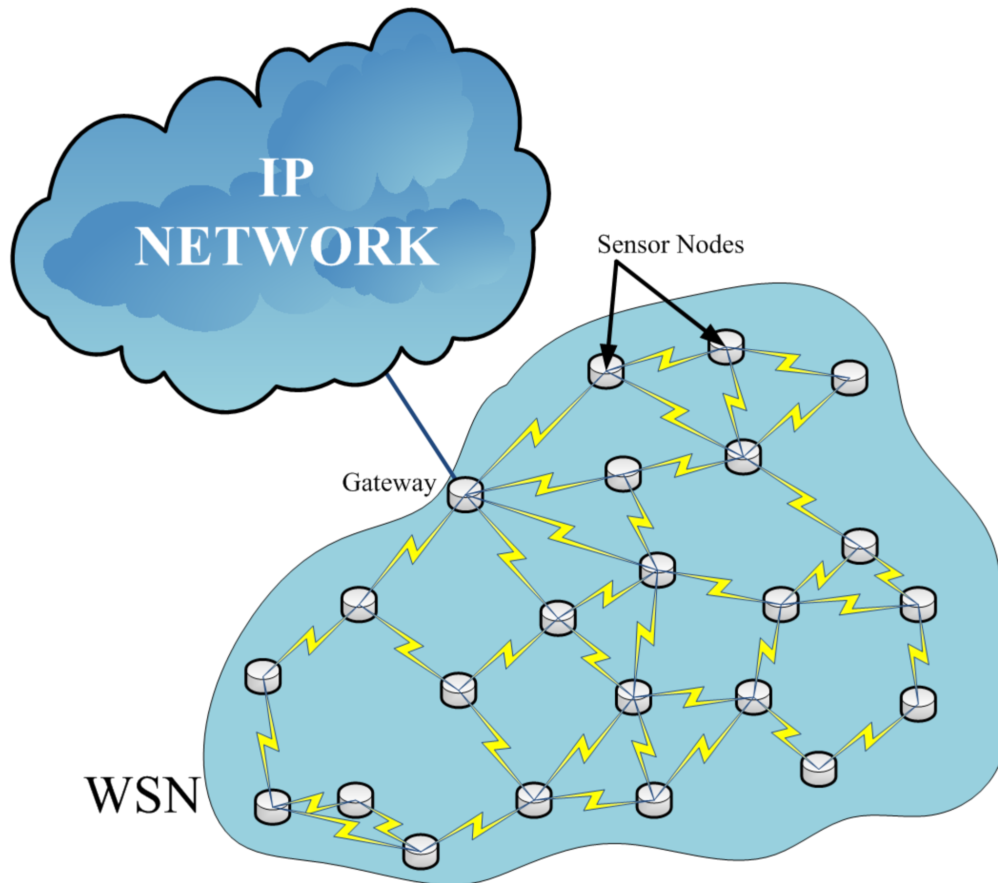


Figure 7.2: This figure represents the network architecture of an approach in which the WSN is directly connected to the Internet, and each WSN node contains the full TCP/IP networking stack. The node directly connected to the IP network communicates wirelessly with the nodes in the WSN within its communication range and forwards information on behalf of IP-based hosts and other wireless sensor nodes. This node is the last hop from the WSN to the IP network.

computation? In one scenario, Pottie and Kaiser observe that 3000 instructions could be executed for the same energy cost of sending a single bit 100 meters by radio [10]. This implies the full networking stack should not be included in WSN nodes due to high energy usage and communication overhead. Rather than employing the full networking stack, data-centric networking protocols route data in the network using named data attributes, eliminating the need for host-specific addressing. Therefore, the only transmitted information is that of sending the actual data, not wasting energy on high per-packet overhead.

A further disadvantage of this approach is that just because each sensor node is addressable does not necessarily ease the task of gathering data the sensors can produce. In this case, IP hosts must be supplied with each individual WSN node IP address it wishes to query for data. There could potentially be many WSN nodes in possibly multiple remote WSNs so this may be an inefficient method of information retrieval, especially considering the wasted energy with TCP retransmission attempts. Moreover, this solution does not lend itself well to specialized and energy-efficient WSN protocols inside the WSN.

7.3 Overlay Approaches

In overlay approaches, gateway nodes are used to interconnect WSNs with IP networks and assign virtual identification information to either IP-based hosts, sensor nodes or both. According to [13], overlay approaches come in two basic forms: sensor network overlay IP network and IP network overlay sensor network. These two approaches employ application-layer gateways through which the WSN is identified and information is passed.

7.3.1 Sensor Network Overlay IP Network

In the sensor network overlay IP network approach, IP-based hosts are required to register with the WSN application-layer gateway node and be assigned a *virtual sensor node ID* by the gateway node. IP-based hosts are assigned a virtual sensor node ID because they are not sensor nodes actually located inside the WSN. Thus, the gateway node must keep track of a mapping from virtual sensor node ID to IP address for information coming from a sensor node destined for a virtual sensor node. If a sensor node wishes to communicate with a registered IP-based host — a virtual sensor node — it simply uses the virtual sensor node ID assigned by the gateway node for communication. Once a packet from a sensor node destined for a virtual sensor node ID reaches the gateway node, the gateway node encapsulates the whole packet into a TCP or UDP and IP packet, including any and all WSN protocol headers (e.g., network-, transport- and application-layer headers), and sends

this new packet to the IP host. An illustration of what happens at the gateway node for this approach is shown in Figure 7.3.

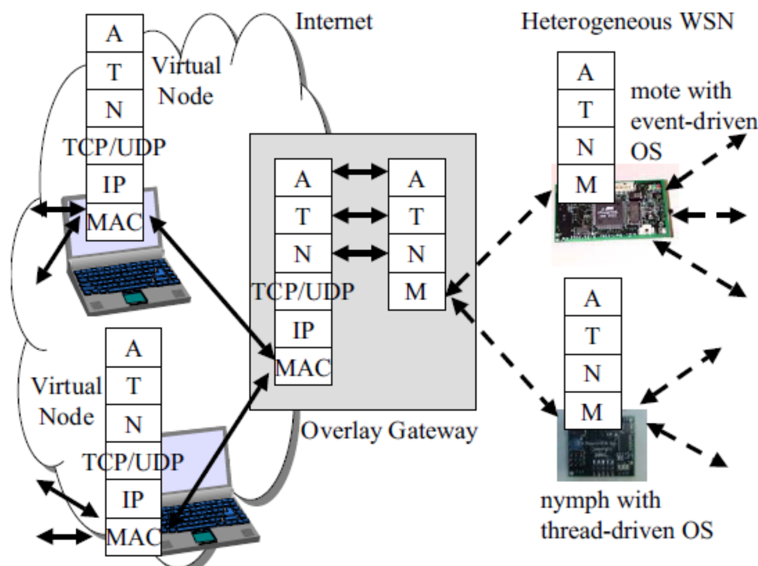


Figure 7.3: This figure, taken from [13], shows the packet translation by the overlay gateway in the sensor network overlay IP network architecture.

The IP-based host communicates with sensor nodes in much the same way, by supplying the sensor node ID to the gateway node. The gateway node then forwards this information onward to the WSN. A major disadvantage of this approach is that it requires the IP-based hosts to be aware of the WSN protocol as it must be able to recognize the encapsulated WSN packet in its entirety. This causes undue complexity in the IP-based hosts and implies this approach may not be well suited for less capable IP-based devices such as PDAs, SmartPhones or other lower energy or hand-held devices.

7.3.2 IP Network Overlay Sensor Network

In this approach, sensor nodes are required to register with the WSN gateway node and are assigned a virtual IP address; however, this is a little misleading as the individual sensors themselves do not actually possess an IP address in the WSN. Sensor nodes are instead assigned a WSN-wide unique standard 16-bit TCP/UDP port number by the gateway node.

IP-based hosts communicate with individual WSN nodes by supplying the IP address of the gateway node and port of the sensor node with which it wishes to communicate. In this way, the gateway node acts as a NAT router by mapping the supplied port number to individual sensor nodes; thus, a requirement of this approach is that the gateway node must keep track of the mapping from TCP/UDP port number to sensor node ID.

This scheme has its issues. Firstly, if the standard 16-bit unassigned TCP/IP port numbers are used to identify individual sensor nodes, only around 16,000 nodes can be uniquely addressed. This may be prohibitive if a large number of sensors are expected to be deployed. This approach also suffers from the protocol overhead attributable to TCP/IP. This approach may also suffer from large routing tables due to the fact that the gateway node must keep track of two different mappings: a sensor node ID-to-IP address map and an IP address-to-sensor node ID map. It is possible to overload the memory of a gateway node if a very large number of sensor nodes and a very large number of IP-based hosts wish to communicate. Also, the gateway node is being given a substantial amount of responsibility in assigning unique identification information to all of the sensor nodes, making the gateway node more complex.

Aside from these issues, there is still the issue of actually gathering the data from each individual sensor node and neither of these two overlay approaches truly simplifies the task of gathering data from WSNs.

7.4 6LoWPAN and IEEE 802.15.4 Standards

6LoWPAN is an acronym of *IPv**6** over **Lo**w power **W**ireless **P**ersonal **A**rea **N**etworks* and is the name of a working group in the Internet area of the Internet Engineering Task Force (IETF). Contributors and the IETF have published several Requests for Comment[1] (RFCs) which were intended to standardize mechanisms that allow IPv6 packets to be sent to and received from IEEE 802.15.4-based networks. IEEE 802.15.4 networks are based

[1] RFCs 4919 and 4944

on low-rate wireless personal area networks (LR-WPANs) and are very well-suited for low-power devices. The IEEE 802.15.4 standard defines the physical layer and media access control for the wireless personal area network, and the proposed 6LoWPAN standard defines encapsulation and header compression mechanisms that allow seamless IP integration over IEEE 802.15.4. In 6LoWPAN, individual sensor nodes are addressable with standard IPv6 IP addresses without the overhead of sending full IP addresses when routing messages inside the WSN. This is because a gateway node connected to an IP network maps full IP addresses into 16-bit node IDs for more efficient bandwidth usage along wireless hops.

This standard, however, is only in its preliminary stages and thus will probably undergo more changes before the final standard is widely available. Like other IP-enabled approaches to interconnecting WSNs, this approach also requires that IP-based hosts know the specific IP addresses of sensor nodes with whom they wish to gather data. Further, the IEEE 802.15.4 standard defines a maximum bandwidth that may be unsuitable for WSN applications requiring larger bandwidths. Moreover, these standards together or separately do not necessarily ease the task of gathering data from sensor nodes in one or more WSNs although it does reduce the amount of wasted energy with respect to transmitting end-point identification information for each packet along every wireless hop.

The following section focuses on methods of actually collecting data from sensor nodes within WSNs rather than on interconnection techniques to enable information to flow from WSNs to IP networks.

7.5 Data Collection from WSNs

Much research has also been done in the actual collection of data from nodes in WSNs in recent years. Different clustering [32, 6, 12], routing [23, 19, 15, 14], data-centric storage [20] and query optimization [22, 16] protocols have been theorized and implemented in the past and are concerned with, for the most part, energy efficiency since most of these ideas have been developed for energy-constrained wireless sensor nodes. Among these ideas, too,

are service-oriented middleware approaches [31, 18]. These techniques and approaches are discussed further in the following subsections only for informational purposes. The middleware-based approach to gathering sensed data from WSNs from IP-based hosts discussed in this thesis could conceivably be built on a number of the following techniques, methods and/or protocols, but, until the writing of this thesis, no definitive API for both WSN sensor nodes nor IP-based hosts has been discussed in the literature.

7.5.1 Clustering, Routing and Data-Centric Storage Protocols

Several clustering, routing and data-centric storage protocols have been discussed in the literature. Most relevant to this thesis are protocols which enable efficient information retrieval. This retrieval of information uses energy resources and, therefore, efficient information retrieval protocols should be used to maximize the lifetime of the system.

As far as clustering protocols are concerned, the most notable contributions are those described in [32, 6, 12]. Clustering protocols achieve energy efficiency primarily by issuing commands to only a subset of sensor nodes at any given time. The authors of [12] propose a Low-Energy Adaptive Clustering Hierarchy (LEACH), a clustering-based protocol that periodically rotates local clusterheads to evenly distribute the energy load among the sensor nodes in the network. LEACH uses localized coordination to enable scalability and robustness for dynamic networks, and incorporates data fusion into the routing protocol to reduce the amount of information that must be transmitted at each hop. The authors of [6] take the ideas contained in LEACH a bit further and develop a power efficient data query processing protocol (PEDQPP). PEDQPP essentially marks the sensor node area with a virtual grid and sensor nodes located within the same cell of the virtual grid form a sensor node cluster. A local clusterhead located within a cell aggregates and stores data. Once a query task is sent to the WSN, a minimum spanning tree is formed with the base station as the root and all the clusterheads in each virtual grid cell as children. The query task floods only along the minimum spanning tree therefore reducing the amount of energy in transmitting

queries. The authors of [32] take a similar approach to PEDQPP except that clusters are constructed online from knowledge of sensor node and query locations only in the query area and only when required to complete a query. This kind of on-demand clustering protocol has been shown to save energy in simulations but requires quite a bit more knowledge of the environment than the other two approaches.

WSN routing protocols take a different approach to achieve energy efficiency by forming paths in ways that cause less energy to be consumed in the routing of information. The authors of [23] describe the collection tree protocol (CTP), one of a family of tree collection protocols which forms a spanning tree between nodes in the sensor network and rooted at some collection entity. Queries and information are routed along this tree to reduce energy consumption at each intermediate node. Cross-layer link estimation is at the core of this approach. Rather than sending probe control packets, the authors use data traffic as active topology probes, requiring control packets to be sent only periodically when the network typology is static. The authors of [19] describe TAG, a Tiny AGgregation scheme based on TinyOS and a spanning tree protocol used to aggregate information at nodes' parents while drawing the information to the tree's root. The authors of [15] take a different approach and present a family of adaptive protocols they dubbed SPIN (Sensor Protocols for Information via Negotiation). In this family of routing protocols, data is named using high-level data descriptors called meta-data. Sensor nodes negotiation this meta-data in order to eliminate the transmission of unnecessary redundant information. The authors of [14] describe a gateway discovery protocol for sensor nodes inside a WSN that wish to send information to an Internet host. The WSN is assumed to have multiple gateway nodes, and sensor nodes begin by sending out special probe packets a single hop, then two hops and then three hops until such time the closest gateway node connected to the Internet is reached. This gateway node then sends a reply back to the sensor node and the sensor node then begins transmitting information to this gateway node to forward to the IP network.

The authors of [20] describe a protocol in which information is distributed over a WSN which acts as a database for sensed information in a manner known as data-centric storage (DCS). When a sensor node detects an event, it decides which location is responsible for keeping the information and the information is automatically sent to a node in that location. The location depends on the type of data sensed and can be computed from a globally-known hash function. Then, when a node requests a particular piece of information, it can find out the location through the use of the same globally-known hash function. Queries are hashed in a similar manner and are sent only to the locations in the WSN where the data can be retrieved. In this way, energy is conserved because only nodes along a path to a query's "answer" are involved in transmission. Queries and data are routed inside the network using greedy perimeter stateless routing (GPSR). However, protocols relying on DCS suffer from the *hot-spot problem* where any localized area in a WSN where relatively more radio transmissions take place diminish the energy of nodes in that area.

7.5.2 Query Optimization

The authors of [16] implement a system called TinyDB that runs in networks of TinyOS-based Berkeley motes. It is based on the TAG scheme as described above and is based on simple and declarative SQL-like query semantics. In [22], the authors similarly use declarative queries posed in a variant of SQL, but focus on *multi-query* optimization in which data and query dissemination are optimized with respect to the multiple submitted queries.

7.5.3 Middleware Solutions

The authors of [31] categorize and summarize several different middleware approaches and highlight some of the associated challenges. The two main approaches for middleware solutions are to provide programming abstractions and programming support. Middleware solutions range from virtual machines where developers write applications in separate, small modules and the system injects and distributes the modules through the network to

databases where the entire WSN is viewed as a virtual database system which can be queried. Most similar to the middleware approach used in this thesis is the programming support abstraction. Another middleware solution can be found in [18]. The authors of [18] describe the use of Global Sensor Network (GSN) middleware which supports the flexible integration and discovery of sensor networks and sensor data, enables fast deployment and addition of new platforms, provides distributed querying, filtering, and combination of sensor data and supports the dynamic adaptation of the system configuration during operation. GSN's central concept is the *virtual sensor abstraction* which enables the user to declaratively specify XML-based deployment descriptors in combination with the possibility to integrate sensor network data through plain SQL queries over local and remote sensor data sources.

These clustering, routing and data-centric storage protocols and query optimization and middleware solutions just discussed in the previous subsections do not alone solve the problem of retrieving information from one or multiple WSNs and delivering that information to interested IP-based hosts. A system overview is given in the next chapter to describe the data-harvesting service provided by the system described in this thesis.

Chapter 8

System Overview

The overall architecture of the system introduced in this thesis plays a very important role. A more natural architecture will ease implementation issues and make the system more scalable and robust. Using the proven existing IP infrastructure is at the heart of the overall network architecture. The Internet already connects billions of computers and users across all continents on earth, so what better way to connect those computers and users to sensed data provided by WSNs? By connecting these WSNs to the Internet, different people from around the world will be able to use sensed information in ways never before imagined.

The overall network architecture is meant to be flexible enough to allow any host around the world connected to the Internet to gather sensed data from one or more WSNs. In Figure 8.1 below, note the many different types of IP-enabled devices that can gather data from one or more WSNs. Many different IP-enabled devices — from PDAs on wireless LANs and SmartPhones with cellular service providers and laptops and desktops on wired and wireless LANs to servers that can act as front-ends to other yet-to-be-discovered services — can all benefit from this all-encompassing architecture. These devices can even access data from more than one WSN located remotely connected to the Internet (Figure 8.2). The Internet can connect these different devices, possibly having different hardware platforms, to facilitate ease of gathering data from WSNs.

A sample query using this architecture is shown below in Figures 8.3 and 8.4. First, an External Agent submits a request to the gateway node with the name of the named data it wishes to consume, shown with a dotted red line in Figure 8.3. The gateway node translates this request into an interest that the Directed Diffusion networking protocol can understand.

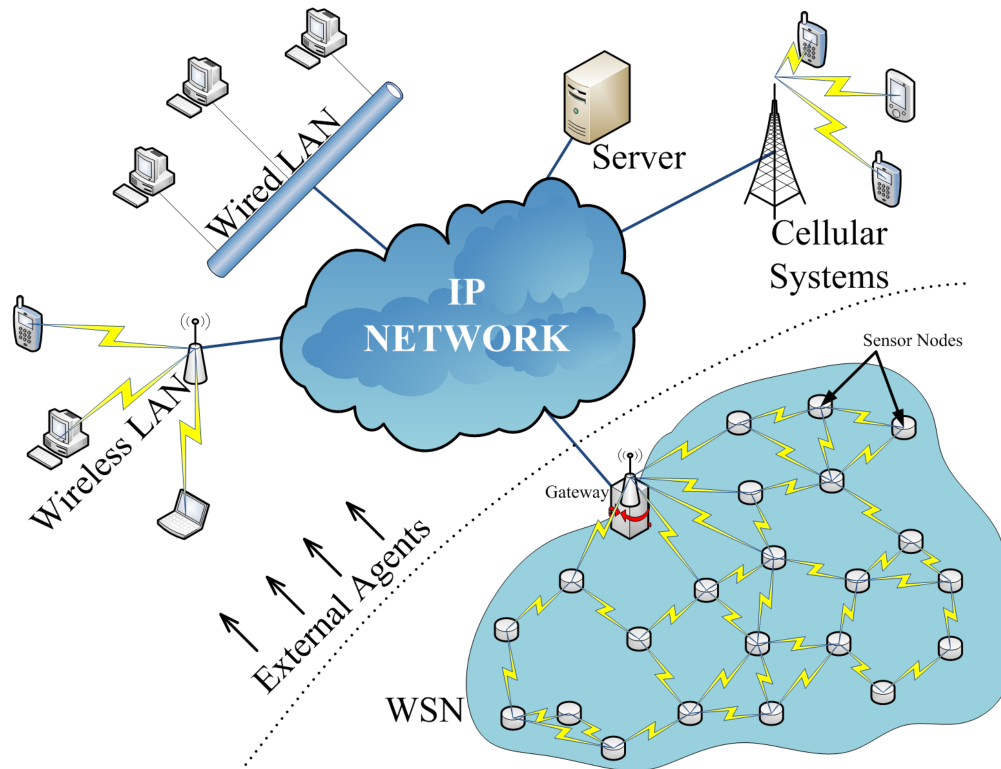


Figure 8.1: This figure is a representative network architecture showing various heterogeneous IP hosts — *External Agents* with respect to the interior of the WSN — and a single WSN. Below and/or to the right of the curved dotted line, a gateway node communicates wirelessly with the nodes in the WSN within its communication range. The gateway node is responsible for forwarding interests from IP hosts to the sensor nodes in the WSN and data from the WSN sensor nodes to IP hosts.

The interest is then flooded to the other nodes in the WSN, shown with a solid green line in Figure 8.3.

All sensor nodes, upon receiving this interest, check to see if it can, in fact, produce data corresponding to the name of the received interest. If a sensor node is capable of producing data corresponding to the interest — these sensor nodes are marked by the lighter yellow color — it then transmits its data back to the gateway node, shown with a solid purple line in Figure 8.4. The gateway node, recognizing that the name of the received data matches an earlier subscription to this named data, forwards the payload of the DD data packet, shown with a dark dotted red line in Figure 8.4, onward to the proper External Agent(s).

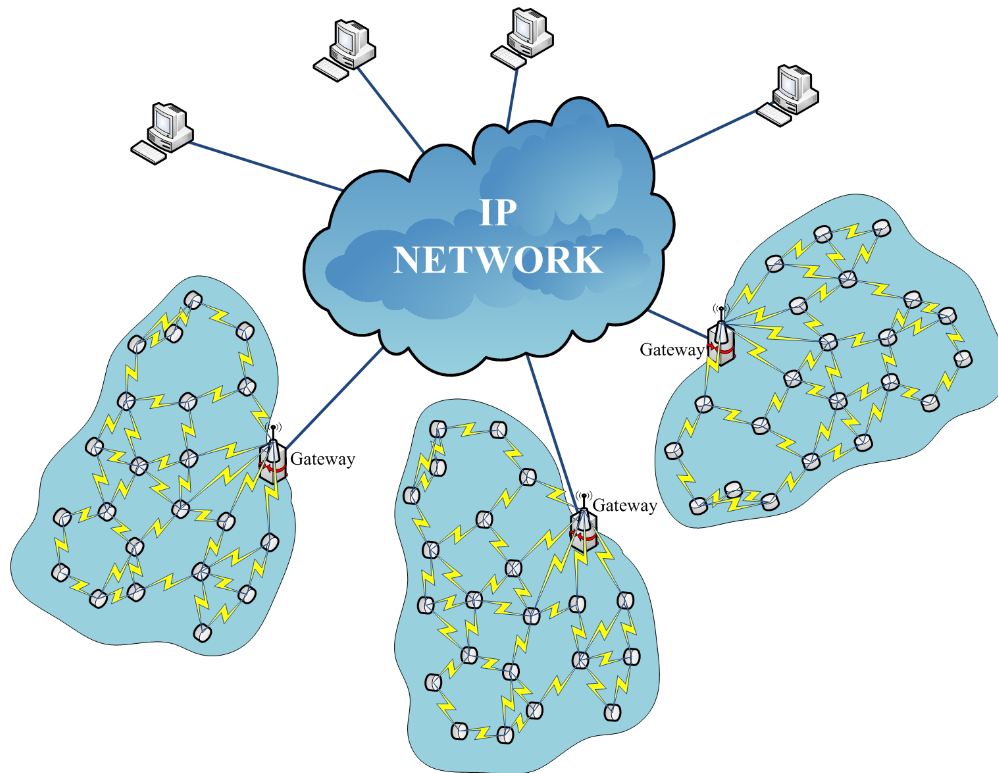


Figure 8.2: This figure is another representative network architecture showing IP hosts and multiple WSNs. The figure is meant to show the possibility that a single IP host can gather data from not just a single WSN, but multiple WSNs using a single, simple interface.

Figure 8.5 shows the various network layers through which requests travel from an IP-based host on the Internet, up and down the IP and diffusion networking protocol stacks on the gateway node, respectively, and finally traveling up the DS-enabled DD network stack of the sensor node. Data traveling from a sensor node to an IP-based host follows the same path but in the reverse direction. It is important to note here that DS does *not* replace the transport layer of the traditional networking stack. As a matter of fact, the Directed Diffusion layer in the sensor nodes in the WSN actually takes care of the data transport and networking layer functionality.

Also important to note here is the fact that the gateway node does not require the use of DS. Only sensor nodes which have running applications which produce data require the use of DS.

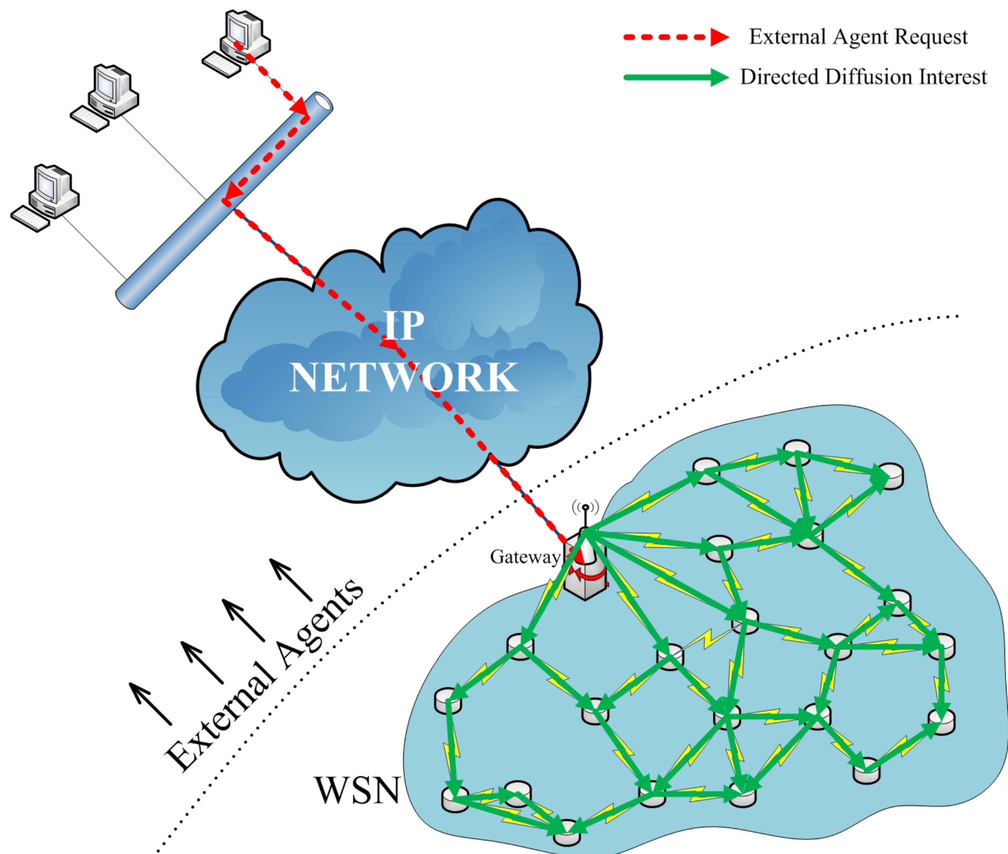


Figure 8.3: This figure shows an IP-based host querying a wireless sensor network through the gateway node situated on the boundary of the WSN. The External Agent's request travels through the IP network — the Internet — and is delivered to the gateway node. The gateway node processes the interest and transmits the interest to the WSN in the form of a Directed Diffusion interest message.

In the sections that follow in this system overview, a discussion is given of Directed Diffusion (DD) and its role in the WSN as the data-centric networking protocol. A brief introduction to Dynamic Services and the different agent roles at play within the overall system is also given.

8.1 Directed Diffusion

This research uses and is enabled by Directed Diffusion (DD), a relatively new data-centric ad-hoc networking protocol capable of robust, scalable and energy-efficient multi-hop communication [5]. It was the brainchild of researchers from the Scalable Coordination

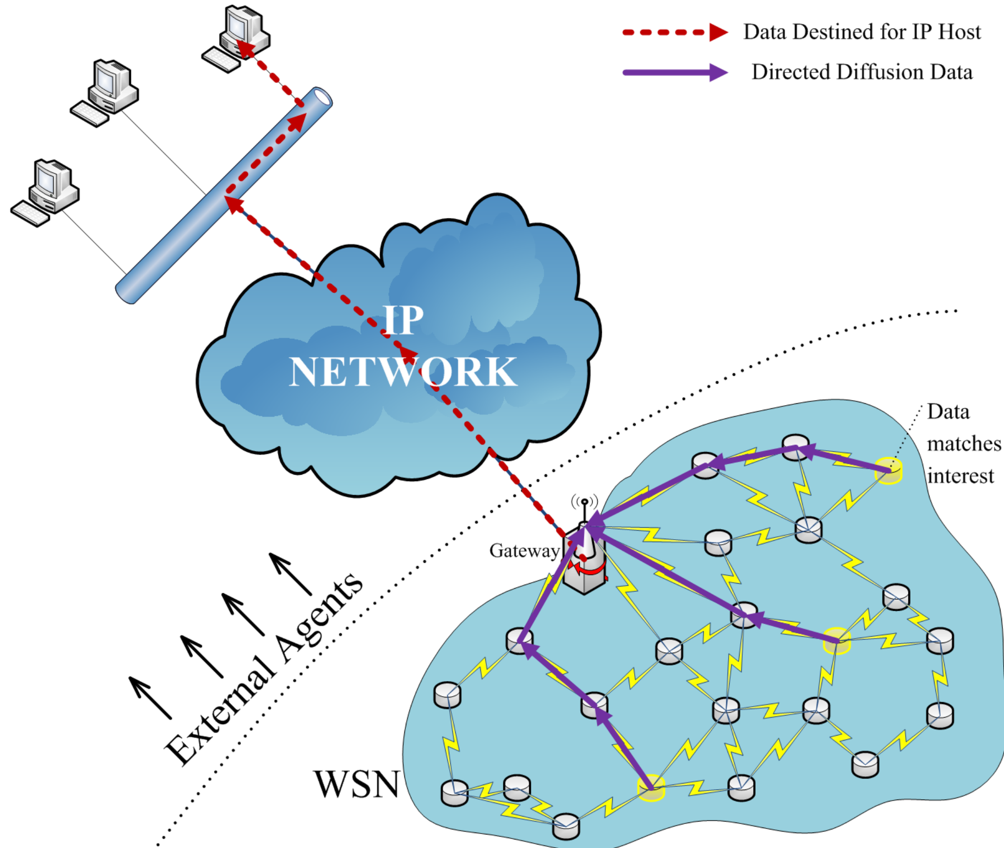


Figure 8.4: This figure shows an IP-based host receiving data from a wireless sensor network via the Internet. Directed diffusion data matching the interest previously sent by the External Agent is transmitted through the Directed Diffusion network back to the gateway node. The gateway then forwards this data back to the External Agent(s) who previously requested the data.

Architecture for Deeply Distributed Systems (SCADDS) Lab at the University of Southern California circa 2000 funded by the Defense Advanced Research Projects Agency (DARPA). The software — the DD networking protocol algorithms — they implemented has gone through a few revisions including adding reliable multi-segment transport (RMST). The version used to construct the solution provided in this thesis is currently the latest version of diffusion, v3.2.0.

An attractive feature of DD, especially in the context of wireless sensor networks, is that broken paths mend themselves. Periodically, exploratory data is flooded throughout the network for this purpose.

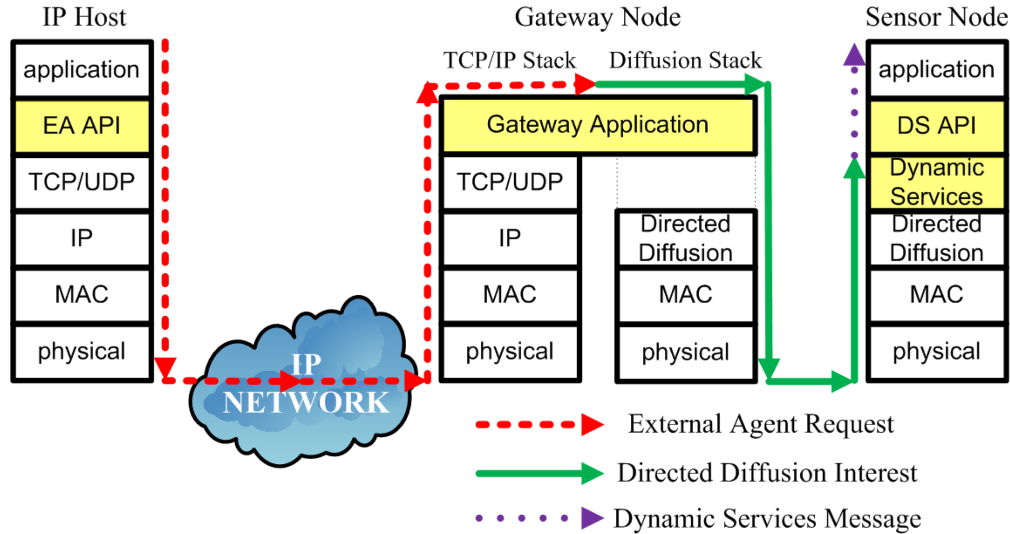


Figure 8.5: This figure shows the various networking layers through which requests travel from an IP-based host, up and down the IP and diffusion networking protocol stacks, respectively, and finally arriving at the application layer of a sensor node. The *API* layer of the IP host on the left is the API presented in Chapter 10. Notice that the EA API, DS API, Dynamic Services and Gateway Application are shaded — these are major contributions of this thesis.

Another attractive feature of DD is that sensor nodes do not generally have a uniquely identifiable address, causing less energy to be used along each wireless hop in communicating node identification information in the TCP or UDP and IP protocols. Instead, *data* is named using attribute-value pairs rather than the individual nodes themselves.

DD was chosen as the data-centric networking protocol to use in this thesis, in part, because it makes routing of the sensed data within the WSN more efficient compared to using TCP/IP on the sensor nodes. The responsibility of passing interests from the gateway node to individual sensor nodes, and subsequent passing of data from sensor nodes back to the gateway node, is given to DD. This makes the task of routing inside the network very simple, convenient and efficient.

Another compelling reason to choose DD is because it keeps routing tables inside the gateway node rather small — smaller than, say, keeping track of both end-points of a conversation as in the other gateway-based approaches mentioned in the Related Work chapter

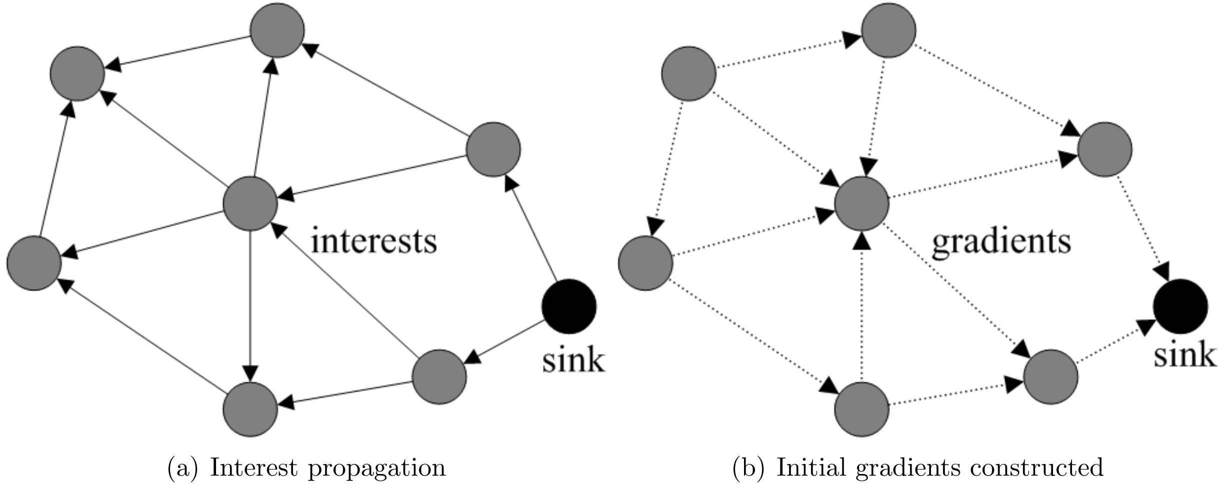
above (Chapter 7). Now, at the gateway node, it is only necessary to keep track of unique interests (subscriptions) and any IP address and port number of IP-based hosts interested in data provided by the WSN.

Because individual sensor nodes are not named in a DD network, there is no overhead in keeping track of or assigning sensor nodes unique IDs nor is there any sensor node energy wasted in transmitting endpoint identification information with each transmitted packet.

8.1.1 Directed Diffusion Basics

The basic DD protocol is very straightforward. When a sensor node has the ability to produce named data, it specifies the name of this named data to the core DD routing algorithm. When the DD core receives an interest for named data that has previously been registered, a callback function — the tasking thread — is invoked to handle the production of data corresponding to this named interest. On the other hand, when the DD core inside the sensor node receives a named interest for named data that has not been previously registered, the DD core will either forward the interest message to its neighbors or drop the interest message altogether. An example DD query is shown below in Figure 8.6 while the data returned from the query is shown in Figure 8.7. Figures 8.6(a)–(b) show a *sink* node, a node interested in some data, along with other sensors in some WSN. When a node is interested in some data, it sends out an interest for that data. The node is then referred to as the "sink" node, the node to which data will be "pulled" if any data matching the interest can be produced by any other sensor node(s) in the WSN receiving the interest. Figure 8.6(a) shows interests being flooded throughout the network, starting from the sink and *diffusing* to all nodes in the sensor network. As these interests are diffused throughout the network, *gradients* are set up along the reverse path of travel of the interests (Figure 8.6(b)).

Figures 8.7(a)–(c) show what happens after the interest is diffused throughout the network. Once a sensor node receives an interest for data it can produce, it begins producing that data. This node is now known as the "source" of data production/transmission. Data



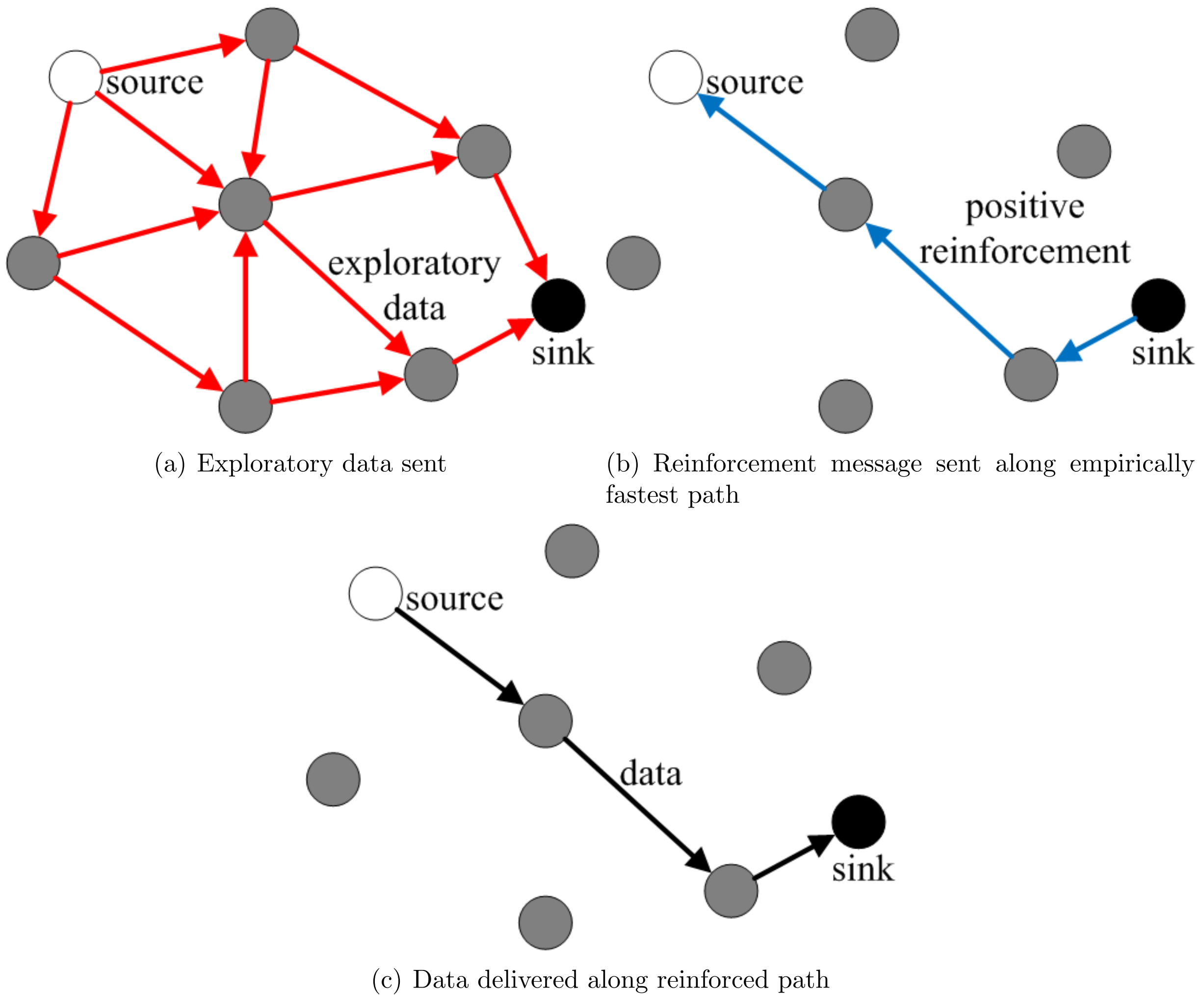
(a) Interest propagation

(b) Initial gradients constructed

Figure 8.6: These figures show the most basic stages of interest propagation in a Directed Diffusion wireless sensor network. Subfigure (a) shows an interest message flowing from the sink — the node which sent out the interest — to the other nodes in the WSN. Subfigure (b) shows the gradients which were set up by that propagating interest.

is forwarded hop-by-hop along multiple gradients and back toward the sink, establishing an empirically fastest path from the sink node to the source of the data, by way of *exploratory data* as shown in Figure 8.7(a). This exploratory data diffuses back across the network across the gradients previously created by the interest's propagation. The empirically fastest path is chosen for reinforcement by the sink for fast data reception of future data packets transmitted from the source. This path is chosen based on which neighbor first transmits data to the sink node. Figure 8.7(b) shows the sink node transmitting a positive reinforcement message to the neighbor from which exploratory data was first received. Subsequently, each node along this path will send positive reinforcement messages along the gradient path until the reinforcement message reaches the source node. From this time onward, data is sent along this positively reinforced path, shown in Figure 8.7(c).

Periodically, however, interests and gradients time out. Therefore, it is necessary for sinks to retransmit the fact they are interested in certain data, reestablishing gradients which have also timed out. This may seem inefficient as far as wireless communication is concerned, but the fact that we are dealing with possibly unreliable sensor nodes and unreliable wireless links means that node and/or link failure or quality degradation may occur. In this case,



(a) Exploratory data sent

(b) Reinforcement message sent along empirically fastest path

(c) Data delivered along reinforced path

Figure 8.7: [These figures are continued from Figure 8.6.] Subfigure (a) shows a node (the node filled with white) who has the capability to produce data matching the interest forwarding that data back to the sink across the gradients. After the exploratory data reaches the sink, the sink sends out a positive reinforcement from the neighbor who delivered the data with shortest delay, shown in subfigure (b), and subsequent nodes along that gradient are positively reinforced. Subfigure (c) shows data being forwarded along the reinforced gradient back toward the sink. In this way, the sink "pulls" data from data sources along gradients towards itself.

gradients and data paths once deemed useful are rendered useless by this dynamic network topology making it necessary to reestablish paths from sources to sinks. The fact that paths mend themselves in an on-demand fashion in a DD network is a very attractive feature of the protocol, especially considering the unreliable nature of WSN nodes and their wireless links.

8.1.2 Directed Diffusion and the Publish/Subscribe Messaging Paradigm

DD follows the publish/subscribe messaging paradigm. In this paradigm, data is divided into various pre-determined classes, and any entity interested in any data must express interest in the data by providing a *key word* corresponding to the name of the pre-determined class of data. Senders (publishers) of data are not given any information of specific receivers (subscribers) to which their data will be sent. Instead, senders simply wait until some entity expresses an interest in some named data and only then do they begin sending out that data. In general, publishers send out data only when they know some subscriber wants, or is interested in, that class of data. In this way, subscribers subscribe only to information they want, and publishers only send data when they know someone wants the data they publish.

Considering Figures 8.6 and 8.7 again, the *sink* node is considered to be a subscriber because it is interested in data from some other entity in the network. The sink node subscribes to some named data by sending out an interest. A sensor node, the *source*, responds to the subscription by publishing data corresponding to the received interest.

8.1.3 Directed Diffusion Sensor Node Application Programming

To understand how DS itself functions, it becomes necessary for the reader to understand how DD functions, and, in particular, it is necessary to understand how DD tasks sensor node applications using the pure DD API.

In any pure DD sensor node application, there are at least two threads of control which run concurrently. The sensor node application programmer creates what is defined as the *tasking thread* — it is specified by the pure DD API that the programmer must create this tasking thread. This tasking thread's only purpose is to wait for an interest message coming from DD. This interest message serves to task the sensor node application. Once this interest message arrives at the tasking thread, a simple integer variable (originally set to zero) which is shared between the two threads is incremented by one to alert the other thread, which is defined as the *main thread*, that the sensor application has been tasked.

All this time, the main thread busily waits checking to determine if the shared variable has been incremented past zero. Once this shared variable has been incremented past zero, the sensor node application should begin its job. Still, the tasking thread continues to run in the background ready to receive what the DD authors call a *disinterest* message. Once the tasking thread receives this disinterest message, it decrements the shared variable. Because the variable is shared, the main thread "sees" this change as soon as it next gains more CPU cycles and continues to busily wait for the shared variable to be incremented past zero once again by an interest message arriving at the tasking thread. In this way, the sensor node application can be tasked, untasked and retasked indefinitely. To see how the main and tasking threads using the pure DD API interact with each other, please see Figure 8.8.

```
sensor_app_main_thread() {
  while (1) {
    if (shared_var > 0)
      /* sensor code */
  }
}
```

```
sensor_app_tasking_thread() {
  if (interest)
     shared_var++;
  if (disinterest)
    shared_var--;
  if (data)
     /* send to app */
 }
```

Figure 8.8: These two threads of execution illustrate — in pseudo-code — how the main and tasking threads are expected to be implemented when using the pure DD API. Notice the shared variable which is incremented and decremented when the tasking thread receives an interest or disinterest, respectively.

In Chapter 9, it will become clear why this detour with the explanation of how pure DD applications function was taken.

8.2 Dynamic Services

Dynamic Services (DS) is an integral part of this solution of information retrieval from the DD WSN, and there is a convenient API that will be detailed in Chapter 9 for sensor node applications to be built on top of DS using DS as a service layer. As previously mentioned, DS is a middleware layer built on top of the DD protocol on wireless sensor nodes to provide the

services necessary to facilitate IP-based information retrieval from sensor node applications built with this type of architecture. Note the difference between the normal DD network stack and the new network stack including the middleware layer DS shown in Figure 8.9. DS is placed between the DD network layer and the application layer. The services provided by the DS middleware layer to the sensor node application programmer allows the sensor node application programmer to ignore the details of the DD networking protocol. The sensor node application programmer need only worry about the neat and clean API to the DD protocol provided by DS. This flexible way to program these WSN nodes is one of the nicest features of the DS service layer.

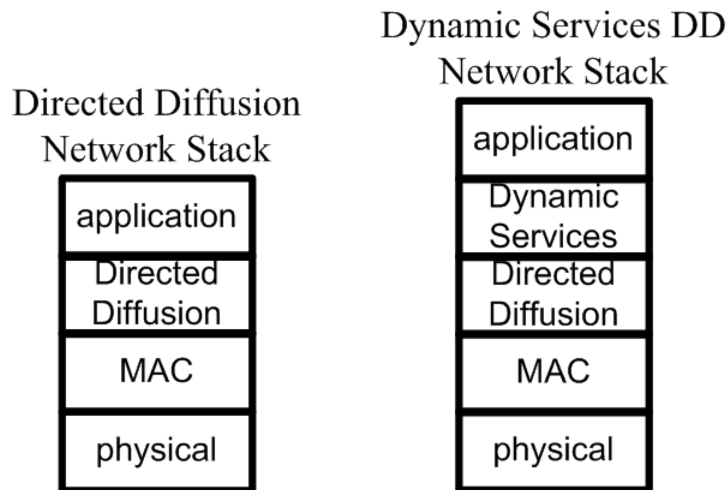


Figure 8.9: This figure shows the difference between the original Directed Diffusion network stack and the modified network stack detailed in this thesis. The major difference is the Dynamic Services service layer included between the application and Directed Diffusion layers.

DS also allows tasking of nodes, specifically for information or data production. Applications register the name of data they can provide with the DS service layer. Until an application receives an interest for this registered named data, it sleeps. Upon receipt of an interest for this registered named data, the sensor node application is awakened by the DS service layer to begin producing the corresponding data. Sensor node applications then send out their produced data, again, through the DS service layer.

8.3 Device Roles

There are three main devices which act in this system. They are External Agents, sensor nodes and the gateway node. These will be introduced and summarized in the sections that follow.

8.3.1 External Agents

An IP-based host that is not a part of the WSN is termed an *External Agent*, or EA. This is because, with respect to the sensor nodes in the WSN and the gateway node that connects the WSN to the Internet, the IP hosts are *external* to the WSN. Notice the curved dotted line in Figures 8.1 and 8.2. This dotted line represents the boundary between External Agents and the WSN. EAs use the API given in Chapter 10 to query and task the sensor nodes in the WSN via the gateway node used to identify the WSN. It is assumed that EAs are full-fledged computers with a full TCP/IP networking stack and can access the Internet or IP network on which the WSN resides.

The EA stack is shown in detail in Figure 8.10. Each EA application running on an EA node possesses its own copy of the EA API. EA applications use this API to submit requests to and receive data from remote WSNs.

8.3.2 Sensor Nodes

The wireless sensor nodes and the data they provide are one of the main components of the system. Sensor nodes must be programmed to recognize interests for which they can supply data. When a sensor node receives an interest for which it can supply data, routines are activated to actually produce the named data. All sensor nodes are assumed to have an antenna capable of wireless communication and sensors that are able to produce some data. Sensor nodes are also assumed to have at least a minimal multitasking operating system (OS) as the middleware layer DS running on each node is its own unique process.

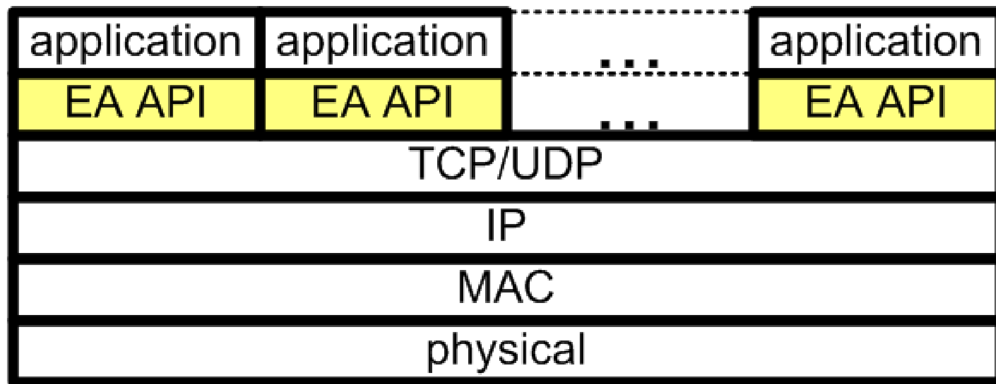


Figure 8.10: This figure shows the networking stack of External Agents in detail. The highlighted area — the EA API (External Agent API) — is a major contribution of this thesis. This layer provides services to the External Agent applications and utilizes the transport services of the lower layers.

The DS-enabled DD networking stack is shown again with more detail in Figure 8.11. Each sensor node application possesses its own version of the DS API. Sensor applications use the API to communicate with the DS service layer, and DS communicates with sensor applications through the DS API.

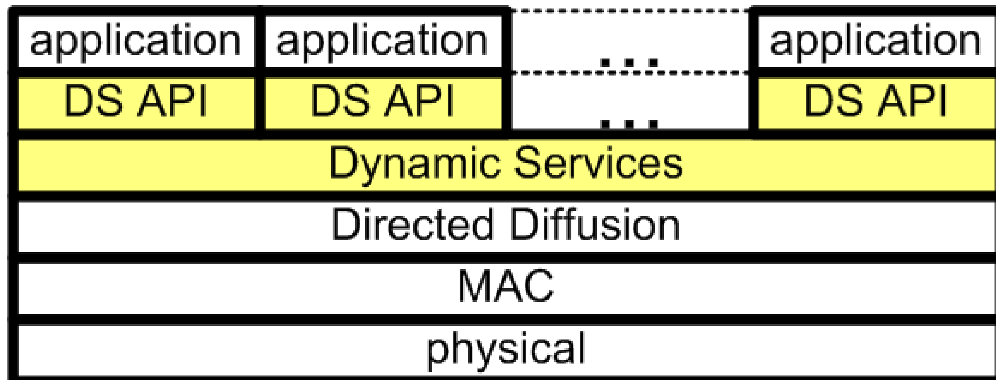


Figure 8.11: This figure shows the Dynamic Services-enabled Directed Diffusion networking stack in detail. The highlighted areas — the DS API and Dynamic Services layers — are a major contribution of this thesis.

8.3.3 Gateway Node

The gateway node sits on the boundary of the WSN and directs traffic coming into and going out of the WSN. Any interest or subscription from an IP-based host first arrives and is processed through the gateway node before the interest enters the WSN. In this way, the gateway node acts on behalf of IP-based hosts in the act of forwarding interest messages onward to the WSN. From the point of view of the WSN, the gateway node acts as a sink for requests originating from External Agents. Therefore, any named data or publication from a sensor node is first routed through the WSN using the DD networking protocol to the gateway node. The gateway node then matches the received data name to named interests from External Agents who have previously subscribed to this named data and forwards the data through IP to the interested IP-based host(s).

The gateway node's networking stack is shown with more detail in Figure 8.12. When the gateway node receives a request from an EA on its IP networking stack, the request travels up the IP-side networking stack to the gateway node application. The gateway application processes this request and converts it into an interest understood by the Directed Diffusion networking protocol. The gateway application then forwards this new packet onward through its Directed Diffusion networking stack onward to the Directed Diffusion network.

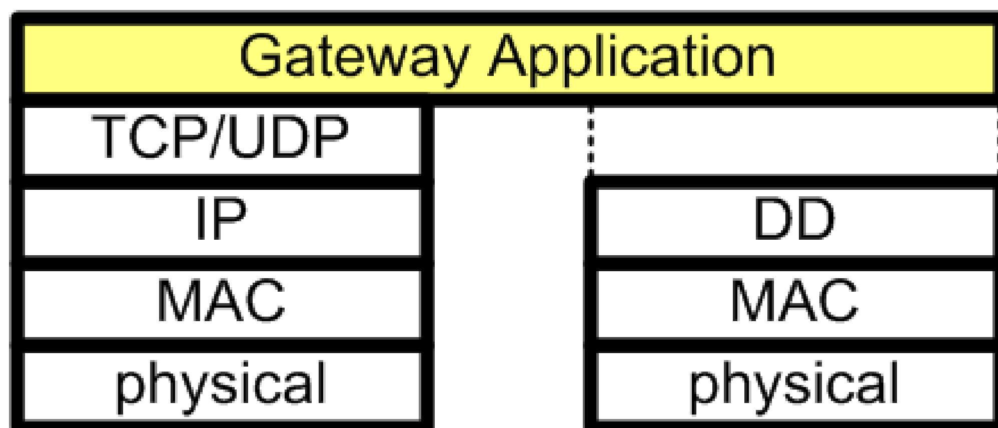


Figure 8.12: This figure shows the networking stacks of the gateway node in detail. The highlighted "Gateway Application" is a major contribution of this thesis.

Chapter 9

Dynamic Services

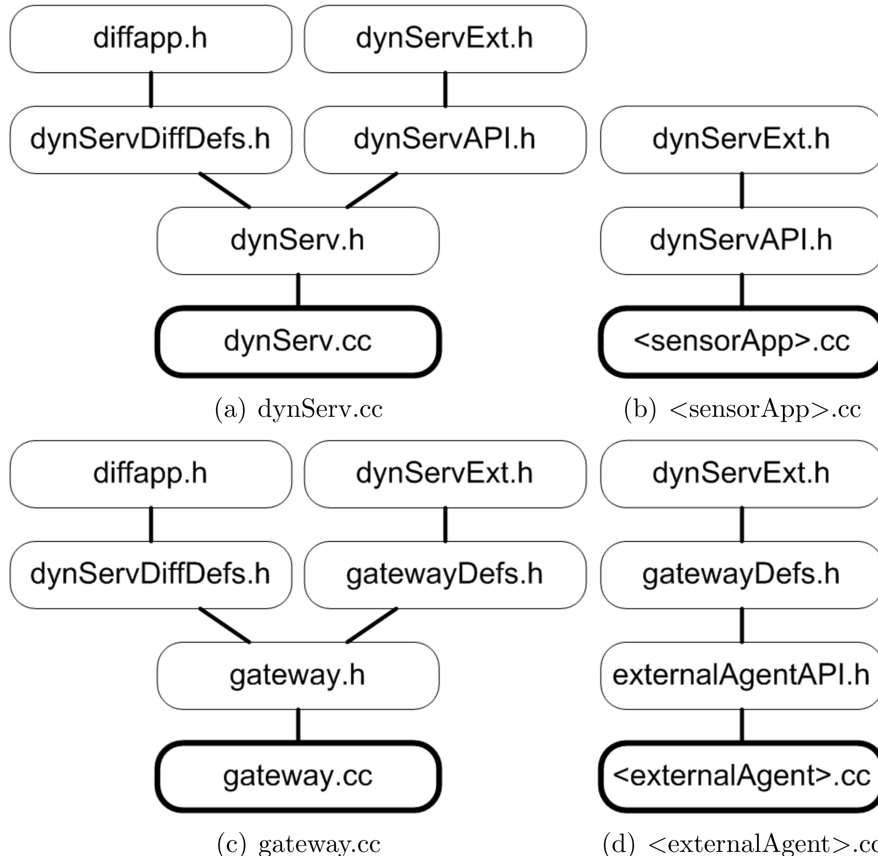
Dynamic Services is a new middleware layer presented in this thesis built on top of DD which makes the programming of individual wireless sensor nodes and data gathering tasks over a DD network easier. In this chapter, details of DS and how it is used in the overall system are given.

There are at least two types of programmers which must be involved in programming this type of WSN querying architecture, the WSN programmer using the DS API and the IP-based application programmer using the External Agent API. IP-based application programmers use the EA API provided in Chapter 10 and sensor network node programmers use a simplified DD API provided as an interface to DS discussed further later in this chapter.

From the point of view of the sensor nodes in the WSN, no sensor node is aware that a host on another network is retrieving information from or tasking a sensor node. IP-based hosts simply "request" that the gateway node submit interests on their behalf. The gateway node maintains a map of this information, a map from named data to IP address, so that once data is received from the WSN by the gateway node, the gateway node can simply look into this map, retrieve the IP address(es) corresponding to the name of the data received, and forward the data packet payload onward to the IP host(s).

9.1 Dynamic Services Include Structure

To understand the rest of this paper, it becomes important to understand the include structure of the software in this system. The include dependency graph is shown in Figure 9.1.



(a) dynServ.cc (b) <sensorApp>.cc

(c) gateway.cc (d) <externalAgent>.cc

Figure 9.1: Subfigures (a)–(d) show the dependence among the different header files of DS, the gateway node and sensor and External Agent applications. Files near the root (bottom) of the graph depend on files nearer to the leaves (top) of the graph. `dynServExt.h` defines each data types' formal structure definition. `gatewayDefs.h` define some useful definitions which enable the External Agent API to submit requests to the gateway node. `dynServDiffDefs.h` define methods which allow DS and the gateway node to extract information from DD packets. `dynServAPI.h` contains function definitions which enable sensor node applications to communicate with and use the services provided by DS.

Figure 9.1(a) represents the include structure of DS. `dynServ.cc` — also known as DS — is, at its heart, a DD application and therefore requires access to the pure DD API. `diffapp.h` is provided by the authors of DD for just this purpose. `diffapp.h` is included in `dynServDiffDefs.h`, a header file which contains certain definitions that allow DS to pack data into and extract data from DD packets. This header file, in turn, is included in `dynServ.h`, the main header file for DS containing mostly class definition information. Extra information used mostly by DS but also needed in the gateway node, External Agents and

sensor nodes is included in `dynServExt.h`. This extra information is stored here so that it is not necessary for External Agents and the gateway and sensor nodes to include all of the class information for DS contained in `dynServ.h`. For instance, `dynServExt.h` contains the actual data structure definitions supported by DS corresponding to the names reserved for named data. This information is needed by both the gateway node and DS in the sensor nodes so that the gateway node can have knowledge of the size of data packets and so that sensor node and External Agent applications can have access to the structure definitions they produce and consume, respectively.

The gateway node's software structure, shown in Figure 9.1(c), also contains `diff-app.h` and `dynServDiffDefs.h` because the gateway node also needs direct access to the DD network as well as information on how to extract data destined for IP-based hosts from DD packets. `dynServDiffDefs.h` and `diffapp.h` are included for this reason. The gateway node, like DS running on the sensor nodes, needs access to `dynServExt.h` because the actual data structure definitions are defined here. This is so that the gateway node knows the size of data packets to transmit to External Agents. `gateway.h` contains mostly class definition information and is supplemented by `gatewayDefs.h`. `gatewayDefs.h` contains information needed by External Agents to enable them to send requests to the gateway node.

Sensor node applications, represented by Figure 9.1(b) should include, at a minimum, the header file `dynServAPI.h`. This header file contains function declarations of the services provided by the DS API and enables sensor node applications to use the services provided by DS. It is also necessary for sensor node applications to be aware of the associated data structure definitions, located in the file `dynServExt.h`, as these are the actual data structure definitions the sensor node applications will fill in order to later transmit in response to interests.

Finally, Figure 9.1(d) represents an External Agent's dependency graph. EA code should include, at a minimum, the header file `externalAgentAPI.h`. External Agents need access to subscribe, unsubscribe and receive functions, and these are defined here. Furthermore,

External Agents also require access to the named data structure definitions provided in `dynServExt.h` since these structures are to be received from the remote WSN(s).

9.2 Inside Dynamic Services

The main idea of DS is to move the complexity out of the individual sensor node applications and into the DS service layer. For sensor node applications to be able to use the services provided by DS, DD and the DS service layer must first be running on the sensor node. Then, sensor node applications can begin running. The DS API functions available to the sensor node application are called within the sensor node application and information is passed from the sensor node application, through a message passing interface (MPI), to the DS service layer. Any information coming from DS to the sensor node application must also pass through this MPI.

As mentioned previously in Chapter 8, DS is a regular DD application and, therefore, contains a tasking thread and a main thread. However, in DS, these threads do not function exactly as the DD authors intended. Put another way, these threads' jobs have been redefined to better suit DS as a service layer. Now, there is no shared variable through which the main thread is tasked by the tasking thread. Rather, each thread now has some job to perform on behalf of the sensor node application or on behalf of the DD network. Please see Table 9.1 for a summary of the redefinition of the tasking and main threads.

To better understand how the two threads inside of DS work, know that communication from the DS API to DS and from DS to the DS API takes place using standard System V Message Queues. The DS API will be introduced in detail shortly in the section which follows. In short, sensor node applications send *register* and *unregister*, *data*, *check publish status* and *subscription* and *unsubscription* messages to DS through the DS API through the *Dynamic Service Message Queue*. *Tasking*, *data* and *check publish status responses* are sent from DS to the *DS API Message Queue* reserved inside of each copy of the DS API. See Figure 9.2 for a simplified diagram of the types of messages which the DS service layer

	Pure DD	**With DS**
Tasking Thread	increments or decrements value of shared variable	tasks sensor node applications through their respective Message Queues when a tasking message is received from DD and passes data to the appropriate sensor node applications through their respective Message Queues when a data message is received from DD
Main Thread	polls shared variable to determine if tasking of the sensor application has taken place	waits at Message Queue for sensor node application requests or data

Table 9.1: This table summarizes the redefinition of the main and tasking threads using the pure DD API versus using the services provided by DS.

and the DS API communicate with each other and through which Message Queues this communication takes place. It is worth mentioning here that the Message Queues used as the MPI for information passing from the DS API to DS and from DS to the DS API are totally hidden from sensor applications. That is, sensor applications are totally unaware that information is being passed to and from DS and the DS API through Message Queues.

The main thread in DS sleeps while awaiting requests sent through its own Message Queue eliminating the polling, or busy waiting, of pure DD sensor node applications. See Figure 9.3 for the pseudo-code corresponding to the main and tasking threads inside DS. Note the difference between the main and tasking threads in DS (Figure 9.3) versus the main and tasking threads in pure DD applications (Figure 8.8).

The *register* and *subscription* messages sent to DS through the DS API through DS's own Message Queue enter information into one or more internal map structures which DS maintains to keep track of sensor node applications. These map structures retain information regarding the data type each sensor node application produces and the data types to which the sensor node applications are subscribed as well as the Message Queue information of each sensor node application through which DS sends responses or data to the sensor application.

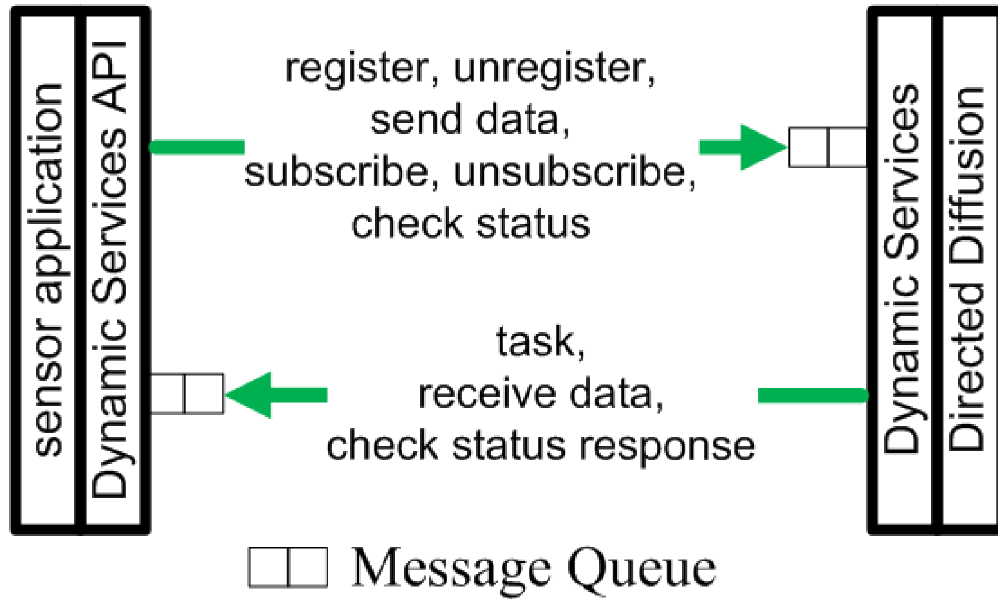


Figure 9.2: This figure shows a simple illustration of which messages are passed between the DS API and the DS Message Queue and from DS to the DS API's Message Queue. Notice that each sensor application's DS API has its own Message Queue which it maintains. The DS API receives tasking, data and check status responses through this Message Queue. There is only one globally accessible Message Queue which is used to input information from the DS API to the Dynamic Service Message Queue.

```
DS_main_thread() {
  while (1) {
    /* block at DS
        Message Queue */
  }
}
```

```
DS_tasking_thread() {
  if (interest)
    /* task app(s) via
        DS API Message Queue */
  if (disinterest)
    /* untask app(s) via
        DS API Message Queue */
  if (data)
    /* send to app(s) via
        DS API Message Queue */
}
```

Figure 9.3: These two threads of execution illustrate — in pseudo-code — how the main and tasking threads are implemented inside Dynamic Services.

After sensor applications are up and running, interests coming from the DD network arrive at the tasking thread in DS. Because the tasking thread also has access to the previously-mentioned internal map structures, DS can use the information contained in these map structures to task the appropriate sensor node application(s). DS looks into these map structures,

locates the Message Queue information of the sensor application(s) and sends *tasking* messages up to sensor node application through the DS API Message Queue, a Message Queue maintained in each sensor application's DS API. For data arriving at the sensor node after a sensor applications submits a subscription request for named data, named data arriving at DS through the tasking thread is sent to the sensor application in a similar manner as tasking information is sent.

9.3 Dynamic Services API Available to Sensor Role

The DS API provides the interface between the application layer and the Directed Diffusion layer on the sensor nodes. There are eight basic functions which were implemented to enable DS to perform its job. The C++ function declarations and explanations are as follows:

- int REGISTER_DATA_TO_PRODUCE(const char *dataType, int *msgQueueID, char *appID, int *DSmsgQueueID);
  - The REGISTER_DATA_TO_PRODUCE function takes in four parameters. The first parameter is the name of the data type this particular sensor node application can produce. The second and third parameters are a returned Message Queue ID — to enable DS to communicate with the sensor application — and an application ID assigned by DS. Process communication between sensor applications and DS take place through System V Message Queues, and these returned values aid in this communication. The last parameter is the Message Queue ID of DS returned from the function to enable quicker access to DS's Message Queue from the sensor application. This function acts to register the sensor node application with DS. A successful call to this function returns one, otherwise it returns negative one. This function effectively tells the DD core that any interests matching this registered data type should not be dropped but instead handed over to the DS tasking

thread so that DS can alert the sensor node application to begin producing data corresponding to the received data type name. No other function from this API should be called without first calling this function.

- int UNREGISTER_DATA_TO_PRODUCE(const char *appID, const char *dataType, int DSmsgQueueID);
  - The UNREGISTER_DATA_TO_PRODUCE function takes in three parameters. The first and last parameters are the returned application ID and DS Message Queue ID from the function REGISTER_DATA_TO_PRODUCE. The second parameter is the data type name the sensor node application no longer wishes to produce. This function tells the diffusion core that the DS tasking thread no longer wants to receive any interest of the data type name specified in the second parameter. In effect, this tells the DD core that this particular sensor node application can not or will not produce any more data corresponding to the data type name given as the second parameter to this function. One is returned on success, and negative one is returned on failure.
- void AWAIT_TASK(int msgQueueID);
  - After the sensor node registers the data type it can produce through calling the REGISTER_DATA_TO_PRODUCE function above, it should call the AWAIT_TASK function with the returned Message Queue ID from the function REGISTER_DATA_TO_PRODUCE to wait for any interests, also called subscriptions, coming from DD that matches its previous registration. When the AWAIT_TASK function returns, the sensor node application programmer should then begin producing its registered type of data and send it out into the DD network using the PUBLISH_DATA function described next.
- int PUBLISH_DATA(int DSmsgQueueID, const Data *data);

- When a sensor node application has been tasked and is ready to send data onto the DD network, it simply calls PUBLISH_DATA with the second parameter pointing to the data it wishes to transmit and with the first parameter being DS's Message Queue ID returned from the call to the function REGISTER_DATA_-TO_PRODUCE. This sends out the data through the DS layer onto the Directed Diffusion network. If the data were published successfully, one is returned; otherwise, negative one is returned. The sensor node application programmer should be careful not to publish data for data type names for which he has not previously registered through a call to the function REGISTER_DATA_TO_PRODUCE.

- int SUBSCRIBE_TO_DATA(const vector<string> *dataTypeNameVector, const char *appID, int DSmsgQueueID);
  - Some sensor node applications need one or more different types of data from other sensor nodes within the same WSN in order to produce their named data. The function SUBSCRIBE_TO_DATA takes care of this functionality of requesting other named data types from inside the WSN. The function takes in a pointer to a C++ STL vector of `string`s, the application ID and DS's Message Queue ID returned from the function REGISTER_DATA_TO_PRODUCE. `string`s are loaded into the vector container in the standard manner using the vector class's member function "push_back." SUBSCRIBE_TO_DATA returns one on success and negative one on failure. This function effectively sends out a named interest onto the DD network for the names of data types packed into the vector structure.

- int UNSUBSCRIBE_FROM_DATA(const vector<string> *dataTypeNameVector, const char *appID, int DSmsgQueueID);
  - Sensor nodes that require data from other sensor nodes within the same WSN may, at some point, not require the named data for which it was previously interested. A call to UNSUBSCRIBE_TO_DATA with the data types packed into the C++

STL vector the application no longer wishes to receive and the application ID returned from the function REGISTER_DATA_TO_PRODUCE will alert the DD core that the sensor node application no longer wishes to receive this named data. DS's Message Queue ID is also required. The function returns one on success and negative one on failure.

- void AWAIT_DATA(int msgQueueID, const Data *data);
  - After a sensor node application calls the SUBSCRIBE_TO_DATA function, it should immediately call the AWAIT_DATA function with the Message Queue ID returned from the function REGISTER_DATA_TO_PRODUCE and a pointer to a `Data` buffer. This blocking function call simply waits for data. When the function returns, the `Data` structure pointed to by the second parameter will be filled with the received data. The function assumes the programmer has reserved sufficient storage space for the returned data type `Data`. The function returns nothing.

- int CHECK_PUBLISH_DATA_STATUS(int DSmsgQueueID, int msgQueueID, const char *appID, const char *dataType);
  - After a node is tasked and begins publishing, or sending, data, the node should periodically check to see if it should continue producing data for interests it received in the past. This is done with the CHECK_PUBLISH_DATA_STATUS function. The programmer should supply CHECK_PUBLISH_DATA_STATUS with DS's Message Queue ID, its own Message Queue ID and application ID and the data type name the programmer wishes to check to see if data production should continue. The function returns one if and only if data should continue to be produced and returns negative one if and only if data production should discontinue to be produced for the specified data type.

9.4 DS API Example Usage

This section gives a simple example usage of the DS API as it is important to understand the DS API for implementing wireless sensor network applications.

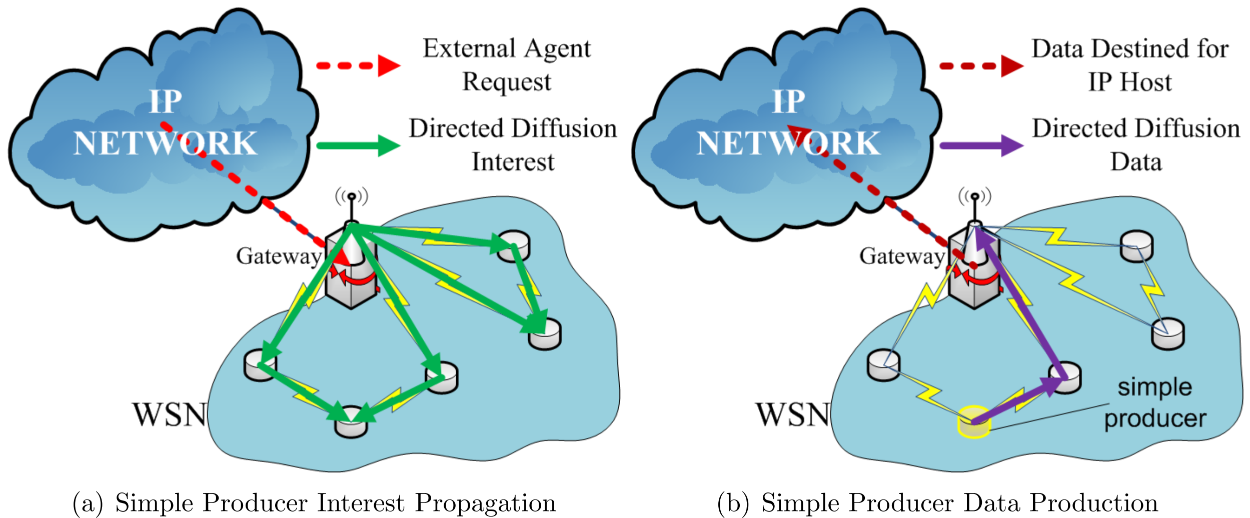
There are two different classes of sensor node applications. One class of sensor node application, when tasked, simply produces the requested named data. These sensor node applications are called *simple producers*. The other class of sensor node applications, however, depends upon other named data types in the local WSN in order to produce its named data. These sensor node applications are called *complex producers*. Example API usage details and examples of each type of sensor application are now given.

9.4.1 Simple Producer API Usage Details & Example

Simple producers, when tasked, simply produce the named data which they were programmed to produce.

When viewed from the network level, the simple producer messaging process looks like that shown in Figures 9.4(a)–(b). See Figure 9.5 to see how messages are communicated between the DS API and the DS service layer for simple producers.

As can be seen in Figure 9.5, sensor applications must first register their intent to publish data. This registration message is passed to the DS service layer through the DS API, reaches the DD core and a registration response message is eventually passed back to the sensor application through the DS service layer and through the DS API. If this registration response indicates a successful registration, the sensor application begins to await tasking. At some point in the future, an interest message flooding the DD network arrives at the sensor node at the DD core. The DD core realizes that a registration for this data type has been received in the past and invokes the tasking thread in DS. The tasking thread in DS then tasks the sensor application through the DS API. The sensor application then begins producing the named data it was programmed to produce. Once the sensor application has data to send out onto the network, it publishes this data through the DS API, through the



(a) Simple Producer Interest Propagation

(b) Simple Producer Data Production

Figure 9.4: These figures show how data production takes place when a sensor node applications known as a simple producer is tasked. Subfigure (a) shows a tasking message originating from the IP network arriving at the gateway node. The gateway node converts this message into a DD interest message which is flooded to the rest of the nodes in the WSN. Subfigure (b) shows the simple producer simply sending or publishing its named data onto the network. This data flows to the gateway node, and the gateway node forwards the data along to the interested IP-based hosts.

DS service layer, and the DD core, at last, actually sends the data out onto the network. A publish success message is propagated up to the sensor application through the DS service layer and through the DS API. Ever so often, the sensor application should check whether it should continue producing the data it was once tasked to produce. The sensor application submits a message to the DS service layer requesting its status to continue producing data. If a disinterest message were received from the DD network before this request is made, the DS service layer replies to the check publication status message with a message indicating that data production should cease. The sensor application should then halt data production and continue to await tasking. The whole simple producer process repeats thus, henceforth.

Now that the reader has been acclimated to what is actually happening at the network layer and between the various layers of the networking stack, this thesis now continues with the example DS API usage for simple producers: At a minimum, all sensor node applications

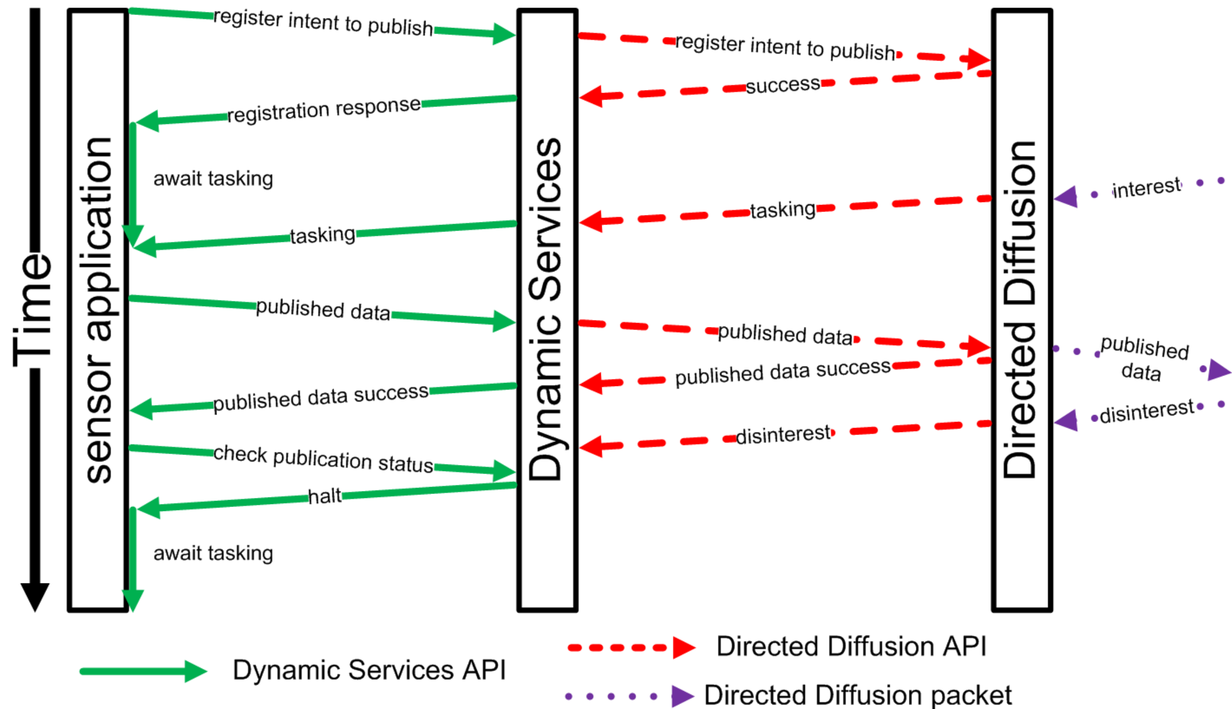


Figure 9.5: This figure shows, in detail, how messages are exchanged for simple producers.

that wish to use the DS API must include the header file `dynServAPI.h` to enable access to the DS API functions described in the previous section.

First, simple producers must declare the data type name it wishes to produce and the `Data` structure it intends to send out onto the DD network. This can be done, simply, using the following:

```
1: char *dataTypeToProduce = "HUMIDITY";
2: Data humidityData;
```

Since there are three return values from the function REGISTER_DATA_TO_PRODUCE, space should be reserved for these values thus:

```
1: int myMsgQueueID;
2: char myAppID[MAX_APP_ID_LENGTH];
3: int DSMsgQueueID;
```

MAX_APP_ID_LENGTH is a constant defined in the included `dynServAPI.h` header file.

Once this has been done, the sensor node application can call REGISTER_DATA_TO_PRODUCE thus:

```
1: if (REGISTER_DATA_TO_PRODUCE(dataTypeToProduce, &myMsgQueueID, myAppID,
       &DSMsgQueueID) != 1) {
2:   /* failure code */
3: }
4: /* success code */
```

Immediately after a successful function call to REGISTER_DATA_TO_PRODUCE, the simple producer should call the function AWAIT_TASK as follows:

```
1: AWAIT_TASK(myMsgQueueID);
```

The simple producer will block at this AWAIT_TASK function, and the whole simple producer process will sleep. The function returns nothing so there is no need to save any return value. Once this blocking function returns, the simple producer should begin producing the named data it registered to produce. Following is an example of named data production and sending the data onto the DD network using the PUBLISH_DATA function:

```
1: strcpy(humidityData.dataType, dataTypeToProduce);
2: humidityData.DataUnion.humidityStruct.dataField1 = /* some data */;
3: humidityData.DataUnion.humidityStruct.dataField2 = /* some data */;
4: /* and so on filling each data field in the Data structure */
5: if (PUBLISH_DATA(DSMsgQueueID, &humidityData) != 1) {
6:   /* failure code */
7: }
```

As previously mentioned, the sensor node application should periodically check with DS to determine if named data production should continue or cease. Recall named data production should cease if and only if -1 is returned and should continue if and only if 1 is returned from the CHECK_PUBLISH_DATA_STATUS function. This can be done as follows:

```
1: if (CHECK_PUBLISH_DATA_STATUS(DSMsgQueueID, myMsgQueueID, myAppID,
      dataTypeToProduce) == -1) {
2:  /* code to halt data production */
3: }
```

If data production should cease, the long-lived sensor node application could return control to the AWAIT_TASK function to await further tasking. Otherwise, the sensor node application could unregister from DS in order to shut down gracefully. This can be done as follows:

```
1: if (UNREGISTER_DATA_TO_PRODUCE(myAppID, dataTypeToProduce, DSMsgQueueID)
       == -1) {
2:   /* failure code */
3: }
4: /* graceful exit code */
```

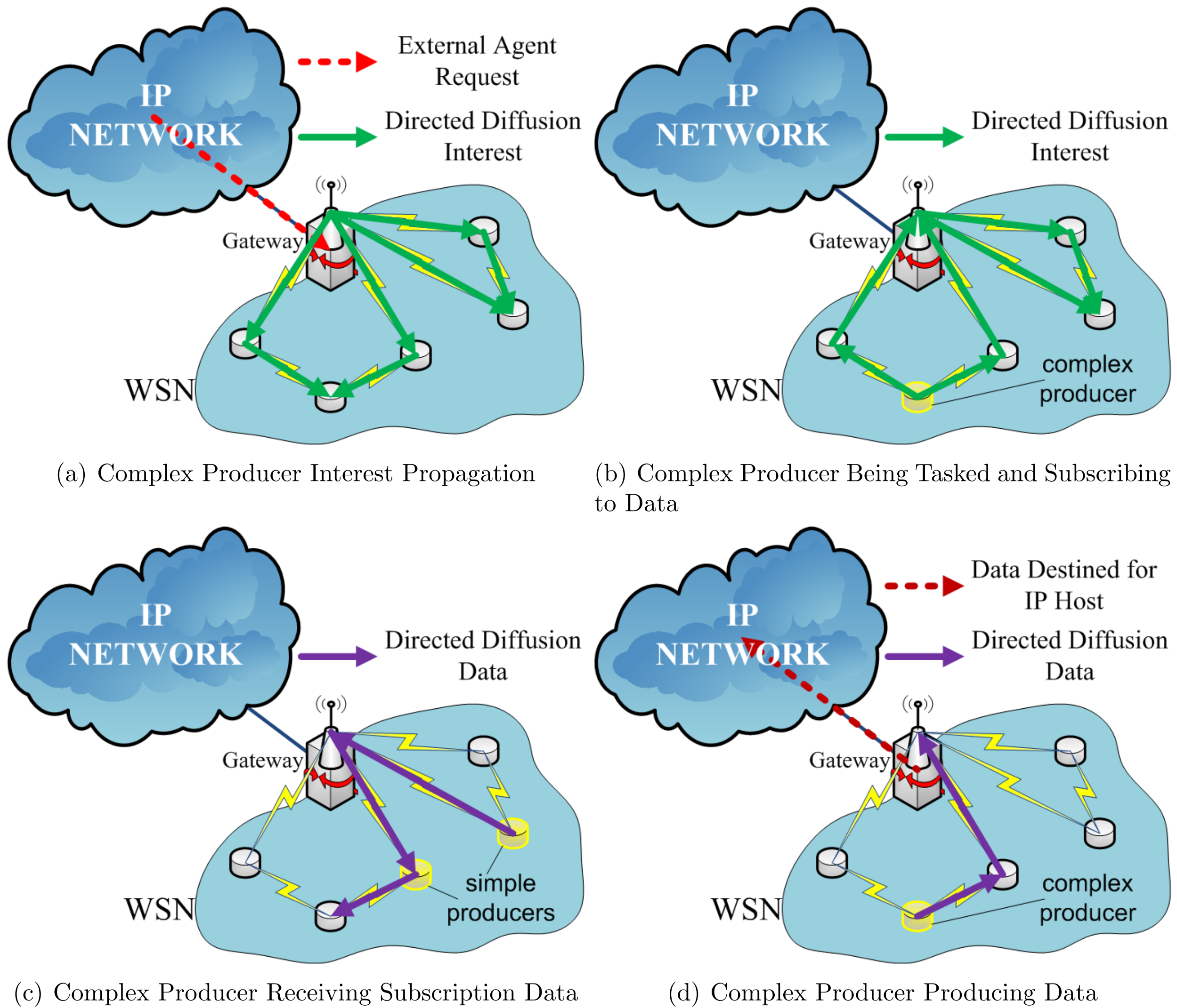
This concludes the simple producer example DS API usage. In the next section, an example of a complex producer's usage of the DS API is given.

9.4.2 Complex Producer API Usage Details & Example

Unlike simple producers, when complex producers are tasked, they require named data produced by other sensor node applications within the local WSN in order to produce their named data.

When viewed from the network level, the complex producer messaging process looks like that shown in Figures 9.6(a)–(d). See Figure 9.7 to see how messages are communicated between the DS API and the DS service layer for complex producers.

As can be seen in Figure 9.7, like simple producers, sensor applications must first register their intent to publish data. This registration message is passed to the DS service layer through the DS API, reaches the DD core and a registration response message is eventually passed back to the sensor application through the DS service layer and through the DS API. If this registration response indicates a successful registration, the sensor application begins to await tasking. At some point in the future, an interest message flooding the DD network arrives at the sensor node at the DD core. The DD core realizes that a registration for this data type has been received in the past and invokes the tasking thread in DS. The tasking thread in DS then tasks the sensor application through the DS API. At this point, the difference between a simple producer and a complex producer become apparent. When a complex producer is tasked, unlike a simple producer, the complex producer subscribes to one or more other data types. This subscription travels through the DS API and into the DS service layer. The DS service layer then submits these subscriptions to the DD core where the subscriptions are turned into DD interest packets which flood the DD network. Upon successfully transmitting this/these interest(s) onto the DD network, a subscription success message is passed up through the DS service layer and up to the sensor application through the DS API. The complex producer then begins to await data from other sensor applications. Eventually, another/other sensor(s) begin(s) to produce the named data for which the complex producer sent (a) subscription(s). The data eventually arrives at the DD core and is passed up to the DS service layer. The DS service layer looks into its internal map structures, determines the sensor applications which subscribed to this data and sends the data to the appropriate sensor applications through the DS API. The data arrives at the sensor application through the DS API. It is advised that after data is received that a check publication status message should be sent to DS; otherwise, the application continues to block



(a) Complex Producer Interest Propagation

(b) Complex Producer Being Tasked and Subscribing to Data

(c) Complex Producer Receiving Subscription Data

(d) Complex Producer Producing Data

Figure 9.6: These figures show how data production takes place when a sensor node applications known as a complex producer is tasked. Subfigure (a) shows a tasking message originating from the IP network arriving at the gateway node. The gateway node converts this message into a DD interest message which is flooded to the rest of the nodes in the WSN. Subfigure (b) shows the tasking of the complex producer. After a complex producer is tasked, it sends out interest messages for other named data types in the local WSN. Other simple (or complex) producers within the WSN are then tasked to produce their named data (Subfigure (c)). Subfigure (d) shows this data flowing from the simple (or complex) producers to the originating complex producer. The complex producer then uses this other named data to produce its own named data. This data flows to the gateway node, and the gateway node forwards the data along to the interested IP-based hosts.

awaiting data or publishing data unnecessarily and also causing other sensor applications to produce data unnecessarily. In the diagram, the first check publication status returns a

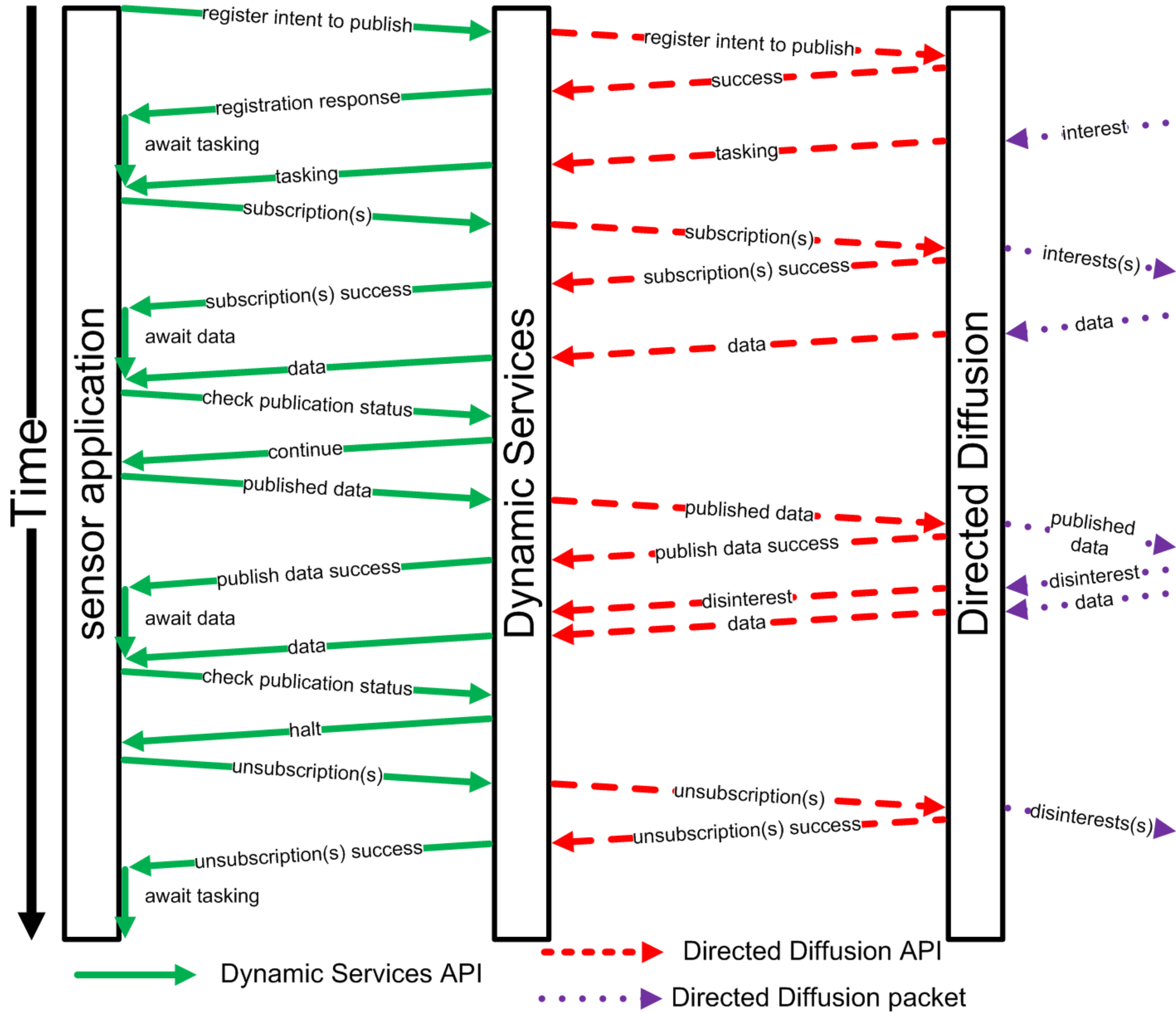


Figure 9.7: This figure shows, in detail, how messages are exchanged for complex producers.

message indicating that data production should continue because no disinterest message has been received to indicate data production should halt. Then, the received data is processed and its own named data is sent through the DS API, through the DS service layer, to DD and eventually out onto the DD network. Just like for a simple producer, the fact that the data was published successfully is returned to the sensor application. The sensor application should then continue to await data. At some point, a disinterest may be received to untask the complex producer. After the next data is received at the complex producer application, the returned check publication status message indicates that data production should halt.

Since this complex producer has previously subscribed to data which is no longer needs, it send (an) unsubscription(s) through the DS API, through the DS service layer and to the DD core. The DD core converts this/these unsubscription request(s) into (a) disinterest(s) and transmits them onto the DD network. Upon successfully transmitting the disinterest(s), the DD core responds to the DS service layer with a success which is propagated up to the sensor application via the DS API. The complex producer should then await further tasking, and the process repeats henceforth.

In the rest of this section, an example usage of a complex producer is given. Again, a complex producer is a sensor node application which requires named data from other sensor node applications in order to produce its registered named data type.

In this example, assume that this complex producer wishes to produce data which is a composite of other sensor node applications' humidity readings. First, complex producers, like simple producers, must declare the data type name it wishes to produce and the `Data` structure it intends to send out onto the DD network. Also, the complex producer needs to reserve space for `Data` returned from the DD network. This can be done using the following:

```
1: char *dataTypeToProduce = "COMPOSITE_HUMIDITY_DATA";
2: Data compositeHumidityData, dataFromDD;
```

Complex producers also need to reserve space for the same three returned values from the function REGISTER_DATA_TO_PRODUCE. This is done thus:

```
1: int myMsgQueueID, DSMsgQueueID;
2: char myAppID[MAX_APP_ID_LENGTH];
```

Again, MAX_APP_ID_LENGTH is a constant defined in the included `dynServAPI.h` header file.

Just like the simple producer, the sensor node application can then call REGISTER_-DATA_TO_PRODUCE thus:

```
1: if (REGISTER_DATA_TO_PRODUCE(dataTypeToProduce, &myMsgQueueID, myAppID,
       &DSMsgQueueID) != 1) {
2:   /* failure code */
3: }
4: /* success code */
```

Immediately after a successful function call to REGISTER_DATA_TO_PRODUCE, the simple producer should call the function AWAIT_TASK as follows:

```
1: AWAIT_TASK(myMsgQueueID);
```

Once this blocking function returns, the complex producer should subscribe to the named data types it requires in order to produce its registered named data type with a call to SUBSCRIBE_TO_DATA. Since this complex producer needs only "HUMIDITY" to do its job, this is the only named data type to which it should subscribe. The subscription is made thus:

```
1: vector<string> dtNameVec;
2: dtNameVec.push_back(string("HUMIDITY"));
3: if (SUBSCRIBE_TO_DATA(&dtNameVec, myAppID, DSMsgQueueID) == -1) {
4:   /* failure code */
5: }
```

Immediately after this function call to SUBSCRIBE_TO_DATA, the complex producer should wait for this data to arrive AWAIT_DATA. This is accomplished thus:

```
1: AWAIT_DATA(myMsgQueueID, &dataFromDD);
```

There is no return value for this function so there is no need to reserve space or check for a returned value. AWAIT_DATA is also a blocking function call so, once the function call has returned, the sensor node application programmer can rest assured there is *some*

data available for processing. At this point, he/she cannot be sure *what* type of named data is available. Recall in the previous section which provides an example for a simple producer, there is a step just before sending where produced data type name is copied into the structure which is to be sent. In receivers, this data type name must be checked to determine which data type has actually been received. If the complex producer only subscribed to one named data type — as in this example — the data type name of the received data may not need to be checked. But, for other complex producers which subscribe to two or more named data types, it becomes necessary to distinguish the received data type with a simple check. This is done as follows:

```
1: if (strcmp(dataFromDD.dataType, "HUMIDITY") == 0) {
2:   /* process the data type accordingly */
3: } else if (strcmp(dataFromDD.dataType, /* some other data type */"")
       == 0) {
4:   /* code to process some other data type */
5: } else {
6:   /* code to handle unknown data type */
7: }
```

Code to handle publishing of data is the same for complex producers as it is for simple producers. See the simple producer example in the previous section.

Just like the simple producer, the sensor node application should periodically check with DS to determine if named data production should continue or cease. Recall named data production should cease if and only if -1 is returned and should continue if and only if 1 is returned from the CHECK_PUBLISH_DATA_STATUS function. This can be done as follows:

```
1: if (CHECK_PUBLISH_DATA_STATUS(DSMsgQueueID, myMsgQueueID, myAppID,
       dataTypeToProduce) == -1) {
```

```
2:   /* code to halt data production */
3: }
```

In the body of the block of code in the preceding paragraph containing `/* code to halt data production */`, should data production discontinue in the complex producer, the sensor node application should unsubscribe from the named data types in this block of code to which it is currently subscribed. This can be accomplished with a call to the function UNSUBSCRIBE_FROM_DATA thus:

```
1: if (UNSUBSCRIBE_FROM_DATA(&dtNameVec, myAppID, DSMsgQueueID) == -1) {
2:   /* failure code */
3: }
4: /* success code */
```

Again, just like in the simple producer, complex producers should, to exit gracefully, unregister with a call to the function UNREGISTER_DATA_TO_PRODUCE. Complex producers should also, before unregistering, be sure to unsubscribe from any named data types for which it currently hold subscriptions.

Chapter 10

External Agent Role

An External Agent is any IP-based host not directly connected to the WSN and can include anything from SmartPhones and PDAs to fully-functioning computers or servers. The key idea is that these devices are decoupled from the WSN protocol and are not a part of the WSN.

One feature of this architecture is that sensor nodes do not expend energy on any sensing or transmission task until an External Agent submits an interest to the WSN. Therefore, External Agents are key players in this network architecture — they actually drive the activities of the WSN sensor nodes by sending requests to the gateway node. To see this, consider the API for sensor node applications given above. Applications first register a data type it can produce and *AWAIT_TASK*ing. That is, sensor node applications sleep until they receive tasking information from some External Agent. This increases the lifetime of the WSN because sensor nodes sleep until they receive tasking information.

10.1 External Agent API

To demonstrate the approach in this thesis actually works from an External Agents' point of view, a simple API was written in C++ that allows any External Agent to be able to retrieve information from a Dynamic Services-enabled Directed Diffusion WSN. External agents must include the header file `externalAgentAPI.h` in order to gain access to these functions.

There are three basic functions, each with a specific job. The function prototypes are shown and described thus:

- int GATEWAY_SUBSCRIBE(const vector<struct sockaddr_in> *addrVec, const vector<string> *dataTypeVec);
  - The GATEWAY_SUBSCRIBE function take in two parameters. The first parameter is a pointer to a standard C++ STL (Standard Template Library) `vector` of `struct sockaddr_in` structures that the programmer has already filled with the appropriate remote WSN gateway IP address and port number information. These `struct sockaddr_in` structures are filled the normal way as for UDP socket programming. The function's second parameter is a pointer to a `vector` of names of data the agent wishes to receive from the remote WSN(s). The standard C++ STL type `string` that is loaded into the `vector` represents the named data type the External Agent is requesting from the DD network. The External Agent packs `string`s into this `vector` just as he would a normal C++ STL `vector`. The `int` that is returned from this function call is actually a UDP socket descriptor through which the gateway node will send data back to the External Agent. The socket descriptor is then used in the RECEIVE_FROM_GATEWAY function to actually receive the data sent from the remote gateway node(s).
- void RECEIVE_FROM_GATEWAY(int sock, const Data *data);
  - After the External Agent application calls GATEWAY_SUBSCRIBE, it then needs to prepare itself to receive any information sent by the gateway node in response to the subscription. The socket descriptor returned from GATEWAY_-SUBSCRIBE is used as the first parameter to RECEIVE_FROM_GATEWAY, and the underlying implementation simply uses this socket descriptor in the familiar *recv* from the standard transport layer's socket library. This function is a blocking call so the External Agent code will block at this function until any data is received from the gateway node, just like the standard TCP/UDP socket blocks until data is received. The second parameter is a pointer to the placeholder where

the data should be stored once this function returns. The function assumes the programmer has reserved sufficient storage space for the returned data.

- int GATEWAY_UNSUBSCRIBE(int sock, const vector<struct sockaddr_in> *addrVec, const vector<string> *dataTypeVec);
  - After the External Agent is no longer interested in a particular type of data from one or more remote WSNs, it uses the socket descriptor returned from GATEWAY_SUBSCRIBE as the first parameter. The External Agent can choose from which remote WSN(s) it wishes to unsubscribe by packing into the second `vector` the `struct sockaddr_in` structures corresponding to those remote WSNs. The data types in which the External Agent is no longer interested is packed into the second `vector`. This makes the gateway node aware of when an External Agent is no longer interested in a particular type of named data. Once no External Agents are interested in a particular type of data, the gateway node sends a disinterest into the WSN so that data production for that named data is no longer wasting energy resources producing named data in which no entity is interested.

10.2 External Agent API Example Usage

The External Agent API was defined in the previous section while this section aims to provide a simple example usage of each of the functions. At a very minimum, the External Agent application programmer must include the header file `externalAgentAPI.h` to access the functions available to External Agents.

First, an External Agent must reserve space for the socket descriptor returned from GATEWAY_SUBSCRIBE and for the `vectors` of `struct sockaddr_in` structures and `strings` which are used to identify remote WSNs and data types, respectively.

```
1: int socketDesc;
2: vector<struct sockaddr_in> remoteGatewaysVec;
```

```
3: vector<string> dtVec;
```

Now, the External Agent must create the `struct sockaddr_in` structure, assign it the information of a remote gateway node and pack it into the correct `vector`. This is done thus:

```
1: struct sockaddr_in gatewayAddr;
2: memset(&gatewayAddr, 0, sizeof(gatewayAddr));
3: gatewayAddr.sin_family = AF_INET;
4: gatewayAddr.sin_addr.s_addr = inet_addr("131.204.142.200");
5: gatewayAddr.sin_port = htons(8989);
6: remoteGatewaysVec.push_back(gatewayAddr);
```

Then, the External Agent must pack `string`s representing named data types it wishes to extract from these remote WSNs and reserve space for data coming from these remote WSNs thus:

```
1: dtVec.push_back(string("HUMIDITY"));
2: Data dataFromWSN;
```

All that remains to subscribe to named data from remote WSNs is to call the function GATEWAY_SUBSCRIBE. This is done as follows:

```
1: if ((socketDesc = GATEWAY_SUBSCRIBE(&remoteGatewaysVec, &dtVec))
       == -1) {
2:   /* failure code */
3: }
4: /* success code */
```

At this point, the IP-based application programmer simply needs to await named data being sent from the remote WSN(s) with a call to the blocking function RECEIVE_FROM_-GATEWAY. There is no return value from this function so there is no need to reserve space or check for any returned value. The function call is made thus:

```
1: RECEIVE_FROM_GATEWAY(socketDesc, &dataFromWSN);
```

When the function returns, named data is available in the `Data` buffer. A programmer can check the type of data returned to be sure it is the named data type he/she expects thus:

```
1: if (strcmp(dataFromWSN.dataType, "HUMIDITY") == 0) {
2:   /* process the named data type accordingly */
3: } else {
4:   /* ignore the unexpected named data type */
5: }
```

When an External Agent application no longer wishes to receive the data types from the remote WSN(s), it simply calls the function GATEWAY_UNSUBSCRIBE as follows:

```
1: if (GATEWAY_UNSUBSCRIBE(socketDesc, &remoteGatewaysVec, &dtVec) == -1) {
2:   /* failure code */
3: }
```

When the External Agent has unsubscribed from all named data types, he can close the socket descriptor and exit gracefully in the normal fashion as follows:

```
1: close(socketDesc);
2: /* code to exit gracefully */
```

This concludes the simple External Agent API example.

Chapter 11

Gateway Role

The gateway role is a very important role in this architecture as it is the entity which physically enables communication from one network to the other. When the gateway node powers on, it simply waits for requests from External Agents in its main thread. When it receives a request from an External Agent, it translates this request into an interest packet that the Directed Diffusion networking protocol can understand. As this interest is diffused throughout the network, gradients are set up along the reverse path of this interest propagation. Named data later produced by a sensor node will traverse the network along these gradients, and the gateway will eventually positively reinforce gradients with empirically shortest delay.

In this way, named data is drawn towards the gateway node for which the gateway node previously sent out interest requests on behalf of External Agents. If the gateway node receives data for which there is no External Agent subscribed, the gateway node simply discards the packet. This could happen if the gateway node receives data from a sensor node that has not yet received the command to stop producing data.

The gateway node software is relatively simple and straightforward. The gateway node is a regular Directed Diffusion application which sits on the boundary of the wireless sensor and IP networks. The gateway node's main thread, upon starting up, prepares an incoming socket on which to receive `Request`s from External Agents. The `Request` structure is totally hidden from External Agents, and therefore, the only thing that External Agents need to know is the IP address and port number of the gateway node, the name of the named interest, the structure of the expected data and how to use the External Agent API. The `Request` data structure is shown thus:

```
1: typedef struct {
2:   RequestType requestType;
3:   char dataType[MAX_DATA_TYPE_NAME_LENGTH];
4: } Request;
```

RequestType on line 2 is either SUBSCRIBE or UNSUBSCRIBE and is defined in gatewayDefs.h thus:

```
1: typedef enum {SUBSCRIBE, UNSUBSCRIBE} RequestType;
```

The External Agent API actually sends this Request to the remote WSN(s). These data structures are never directly manipulated by any External Agent.

When a Request is received from the an External Agent in the gateway node's main thread, an entry is added to a local map structure. This map structure contains the IP address and port number on which the External Agent is awaiting named data. When named data arrives at the gateway node from the WSN, the DD core triggers the gateway node's tasking thread. However, the tasking thread's role in the gateway node has been redefined. Rather than using the tasking thread for tasking, the gateway node's tasking thread looks into the IP address-port number map structure to determine to which External Agent(s) to forward this named data. If any External Agents are found, the named data is forwarded to these External Agents accordingly. If no External Agents are found which has previously subscribed to this named data, the tasking thread completes dropping the received DD packet.

Chapter 12

Integrating the WSN with the IP Network

So far in this thesis a discussion has only been given for each role separately. In this chapter, the reader will be taken on a journey from building a sensor node and IP-based application using the different roles' respective APIs to retrieving named information from a WSN from an IP-based host.

12.1 Building a Sensor Network Application

First, sensor network applications should be built. Only then can an External Agent query the network. When building a sensor network node application, one must first consider what data the sensor node will provide. Let us assume, for the sake of this example application, that a sensor node application wishes to aggregate the temperatures received from three sensor nodes. This sensor node application is a complex producer since it needs named data from other sensor nodes in order to perform its job. Inside DS, and specifically `dynServExt.h`, we need to define the actual data structure of this aggregated named data. For this sensor node application, let us assume that the data structure needs a latitude and longitude position, a sensor node ID and the actual aggregated temperature reading. Therefore, in `dynServExt.h`, a new data structure should be defined for this new named Directed Diffusion data type. In C++, the result should look something like this:

```
1: typedef struct {
2:   unsigned short nodeID;
3:   unsigned long latitude;
4:   unsigned long longitude;
```

```
5:   signed short temperature;
6: } AGGTEMP_struct_;
```

This example assumes there is already a data structure corresponding to "TEMPERATURE" data inside the `DataUnion` which will be produced by other sensor nodes. The "TEMPERATURE" data structure is defined thus:

```
1: typedef struct {
2:   signed short temperature;
3: } TEMPERATURE_struct_;
```

After creating the new structure definition, the new structure should be added to the `DataUnion` union in the `Data` structure along with possibly many more named data type structure definitions shown thus:

```
1: typedef struct {
2:   DataType type;
3:   union {
4:     TEMPERATURE_struct_ TEMPERATURE_struct
5:     AGGTEMP_struct_ AGGTEMP_struct;
6:     ...
7:   } DataUnion;
8: } Data;
```

All of the new named data type structure definitions should be added to this `DataUnion` to decrease complexity in DS and in the gateway node.

At this point, a new sensor node application can be built to produce this new data type.

The sensor node applications should include, at a minimum, the header file `dynServAPI.h`. Space should be reserved for the sensor application's Message Queue ID and application ID, DS's Message Queue ID, and the received and outgoing `Data`. It is also good

programming practice to reserve space for the name of the data type produced and consumed. This can be done thus:

```
1: #include "dynServAPI.h"
2: int myMsgQueueID, dsMsgQueueID;
3: char myAppID[MAX_APP_ID_LENGTH];
4: Data temperatureData, aggTemperatureData;
5: char *toProduce = "AGGTEMP";
6: char *toConsume = "TEMPERATURE";
```

Next, we get into the main coding section of the sensor node application where registration takes place. Again, for continuity, registration takes place like this:

```
1: if (REGISTER_DATA_TO_PRODUCE(toProduce, &myMsgQueueID, myAppID,
        &dsMsgQueueID) == -1) {
2:    /* failure code */
3: }
```

Since it is desirable to have this application continually run on the sensor nodes long-term in case there are any External Agents which want to gather data from this application, an infinite loop situation should be achieved to enable this. The body of this infinite loop is where most all of the controlling and named data production action takes place. Inside this loop, the application will await tasking, subscribe to the named data "TEMPERATURE," check its publish data status, aggregate the temperatures and fill the *aggTemperatureData* `Data` structure, publish the named data "AGGTEMP" and eventually unsubscribe from data if data production should cease. This is all done as follows:

```
 1: aggTemperatureData.DataUnion.AGGTEMP_struct.nodeID = getMyID();
 2: aggTemperatureData.DataUnion.AGGTEMP_struct.latitude = getMyLat();
 3: aggTemperatureData.DataUnion.AGGTEMP_struct.longitude = getMyLong();
```

```
 4: while (1) {
 5:   AWAIT_TASK(myMsgQueueID);
 6:   vector<string> vecOfSubs;
 7:   vecOfSubs.push back(string(toConsume));
 8:   SUBSCRIBE_TO_DATA(&vecOfSubs, dsMsgQueueID, myAppID);
 9:   vector<short> threeTemperatures;
10:   bool keepProducing = true;
11:   while (keepProducing == true) {
12:     AWAIT DATA(myMsgQueueID, &temperatureData);
13:       if (CHECK_PUBLISH_DATA_STATUS(dsMsgQueueID, myMsgQueueID,
            myAppID, toProduce) == -1) {
14:         keepProducing = false;
15:         UNSUBSCRIBE_FROM_DATA(&vecOfSubs, dsMsgQueueID, myAppID);
16:       } else {
17:         if (strcmp(temperatureData.dataType, toConsume) == 0) {
18:           threeTemperatures.push_back(temperatureData.DataUnion.
                TEMPERATURE.temperature);
19:           if (threeTemperatures.size() == 3) {
20:             short avgTemp = /* average the three temperatures */;
21:             threeTemperatures.clear();
22:             aggTemperatureData.DataUnion.AGGTEMP_struct.
                  temperature = avgTemp;
23:             PUBLISH_DATA(dsMsgQueueID, &aggTemperatureData);
24:           }
25:         }
26:       }
27:     }
```

```
28:    }
```

Notice after the flag `keepProducing` is set to `false` in line 14 after CHECK_PUBLISH_DATA_STATUS returns -1 on line 13, the application unsubscribes from the named data "TEMPERATURE" to which it was previously subscribed on line 15. Then, the test in the `while`-loop on line 11 fails, and control returns to AWAIT_TASK on line 5 so that future tasking of the application can occur.

12.2 Building an External Agent Application & Performing a Query

Now that a sensor node application has been built, an External Agent application can be built to take advantage of the named data-gathering service provided by the remote WSN.

Building an IP-based application is even simpler than building a sensor node application, but like the sensor node application, the IP-based application programmer must also include a header file, namely `externalAgentAPI.h`.

As previously mentioned, External Agents must some how know the IP address and port number of remote WSN gateway nodes. Let us assume, for the sake of this example, that the remote WSN gateway IP address and port number are `131.204.142.200` and `8989`, respectively. The External Agent application programmer can now begin the task of building the External Agent code using the External Agent API.

First, the application programmer must create an integer socket descriptor representing the socket through which network communication will take place between the External Agent and the gateway node. Then, the application must prepare the data types it wishes to consume from the remote WSN and pack them into the vector taken as a parameter to GATEWAY_SUBSCRIBE.

Next, the `vector` of `struct sockaddr_in` structures and the `struct sockaddr_in` structure itself should be defined and then filled with the IP address and port number of the remote gateway node. Then, this structure should be packed into the `vector`.

The subscription should then be transmitted to the remote WSNs and the application should reserve space for and begin waiting for data from the WSNs. Awaiting for named data should take place in a loop to continue checking the socket for named data that has arrived.

The process is shown thus:

```
 1: int sock;
 2: char *subscribeData = "AGGTEMP";
 3: vector<string> dtVec;
 4: vec.push_back(string(subscribeData));
 5: vector<struct sockaddr_in> addrVec;
 6: struct sockaddr_in gatewayAddr;
 7: memset(&gatewayAddr, 0, sizeof(gatewayAddr));
 8: gatewayAddr.sin_family = AF_INET;
 9: gatewayAddr.sin_addr.s_addr = inet_addr("131.204.142.200");
10: gatewayAddr.sin_port = htons(8989);
11: remoteGatewaysVec.push_back(gatewayAddr);
12: sock = GATEWAY_SUBSCRIBE(&remoteGatewaysVec, &vec);
13: Data dataPkt;
14: while (1 /* or some other halting criteria */) {
15:   RECEIVE_FROM_GATEWAY(sock, &dataPkt);
16:   if (strcmp(dataPkt.dataType, subscribeData) == 0) {
17:     /* accept and process data */
18:   } else {
19:      /* drop data */
20:   }
21:   if (/*stopping criteria reached*/)
22:     break;
```

```
23: }
24: GATEWAY_UNSUBSCRIBE(sock, &remoteGatewaysVec, &dtVec);
25:  /* code to exit gracefully */
```

Chapter 13

PC104 Testbed & Case Studies

This chapter presents the hardware and software specifications of the PC104 testbed used as the sensor nodes in the WSN and details two sample applications already implemented which use these APIs to show the utility of this approach.

13.1 PC104 Testbed Specifications

In the current version of the PC104-based WSN, we use three PC104-compliant modules, the main CPU module, the PCMCIA module and the Power Supply module.

13.1.1 Hardware

The PC104 CPU module (Figure 13.1), PFM-550S, is manufactured by Aaeon. It has a 533MHz VIA Mark processor and has features such as 10/100Base-TX Fast Ethernet port, one RS-232 port and one RS-232/485 port, four USB 1.1 ports, a SDRAM-SODIMM socket for up to 512 megabytes of RAM and supports type I compact flash cards. It supports 36-bit TL and 18/36-bit dual LVDS LCD panel, has a watchdog timer and fully supports ISA. It is also fanless with an operating temperature of 0 to +60 degrees Celsius. It requires +5V for operation. More information can be found online at http://www.tri-m.com/products/aaeon/pfm550s.html.

The PCMCIA module (Figure 13.2) PCM-3115C, is manufactured by Aaeon. It is a 2-slot PCMCIA module which supports two Type I/II cards or one Type III card. It complies with PCMCIA v2.1 and JEIDA v4.2. It has a 16-bit data bus and a busy status LED. It requires +5V for operation. More information can be found online at http://www.tri-m.com/products/aaeon/pcm3115c.html.

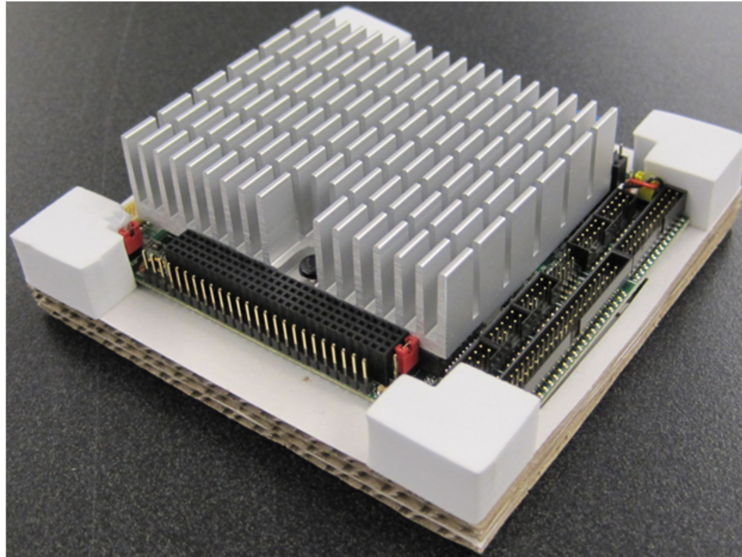


Figure 13.1: This figure shows a view of the PC104 CPU module from the top.

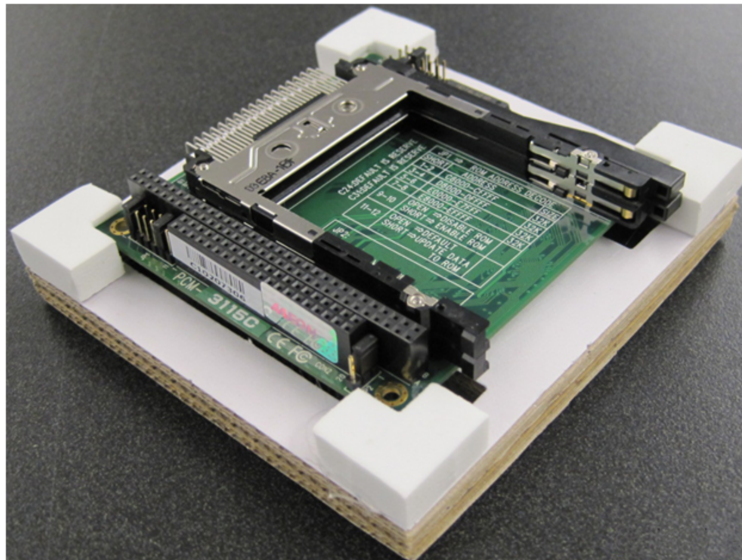


Figure 13.2: This figure shows a view of the PC104 PCMCIA module from the top.

The Power Supply module (Figure 13.3), PFM-P13DW2, is manufactured by Aaeon. It has an input range of +7V to +30V and an output of +5V and +12V. It is used to intake +12V from the AC power supply or battery and convert it into the +5V used by the CPU and PCMCIA modules.

The remaining hardware items described are more commonplace items.

Shown in Figure 13.4 is the Orinoco Gold wireless PCMCIA LAN (Local Area Network) card we use. The wireless LAN card is a 2.4 GHz radio which supports the four IEEE 802.11 High-Speed compliant speeds 11Mb/s, 5.5 Mb/s, 2 Mb/s and 1 Mb/s within the IEEE 802.11 standard for wireless LANs.

Shown in Figure 13.5 is a 2.4-2.5 GHz omnidirectional external antenna used to boost the wireless signal with a +5dBi gain and a 60" cable.

A standard type I compact flash card is shown in Figure 13.6.

The RAM used is a 256MB, 144 pin Synchronous Dynamic RAM high-density memory module at 133 Mhz. Its technical specifications can be found at http://www.transcendusa.com/Support/DLCenter/Datasheet/TS32MSS64V6G_6755.pdf. Please see Figure 13.7 for an image.

A plastic USB microphone, manufactured by Sound Professionals, is a mono, high sensitivity, omnidirectional microphone with headphone amplifier. Its dimensions measure 1.5"x1.0"x0.25", and it can detect frequencies from 20-20,000 Hz. More information can be found online at http://www.soundprofessionals.com/cgi-bin/gold/item/SP-USB-MIC-1. Please see Figure 13.8 for an image.

The AC Power Supply, shown in Figure 13.9, is manufactured by Sunny Computer Technology. It has an input range of 100-240V and an output of +12V. Accompanying the AC Power Supply is the power input connector used to supply power to the PC104 Power Supply module. The +12V AC Power Supply or the +12V battery is connected to the power input connector. We solder wire onto the power input connector here in the lab

Figure 13.3: This figure shows a view of the PC104 Power Supply module from the top.

Figure 13.4: This figure shows a view of the Orinoco Gold wireless PCMCIA LAN card.

Figure 13.5: This figure shows a view of the Lucent omnidirectional external antenna used to boost the signal range of the wireless sensor nodes.

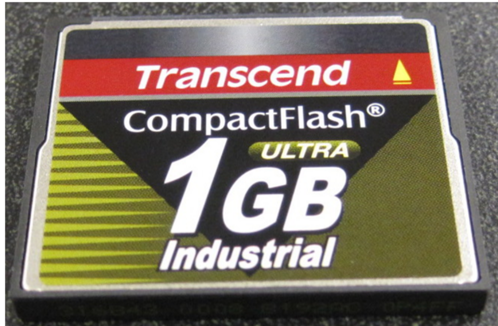


Figure 13.6: This figure shows a standard compact flash card which is the main storage of the PC104 sensor node.

Figure 13.7: This figure shows the 256MB, 144 pin Synchronous Dynamic RAM high-density memory module which is the RAM of the PC104 sensor node.

to connect the wire to the power input connector. For more information, please refer to http://www.sunny-euro.com/HTML/PRODUCTS/POWERSPL/SYS1183UP.html online.

The battery, shown in Figure 13.10, is used to power the PC104 when away from the lab and outputs +12V at 5.0 amp hr. It is sealed and rechargeable. For more information, please refer to http://www.power-sonic.com/site/doc/prod/86.pdf online.

A completed PC104 assembly can be seen in Figure 13.11 and in its protective casing in Figure 13.12. A closeup of the completed PC104 assembly can be seen in Figure 13.13.

13.1.2 Operating System

The PC104 sensor nodes use the Linux-based operating system Slax, v6.0.7, to run all sensor node application software. We chose Slax to use on the PC104 testbed, in part, because it is very small. We removed the GUI (Graphical User Interface) to further save space on the compact flash card. All software used in the testbed, including the operating system, uses around 75MB.

13.2 Sample Case Studies

In this section, two different External Agent applications are discussed which use data supplied by two related wireless sensor network applications in different ways. First, a description will be given of the WSN applications in sufficient detail so the reader can fully understand the basics of the application from the WSN sensor applications to the External Agent applications that use the WSN named data.

Research has shown that algorithms exist for determining the location (at a particular point in time), speed and direction of movement of a target which emits acoustic sound waves traveling through an array of acoustic sensors. The research was first mathematically formalized for flying airplanes in the more general three-dimensional scenario by F. M. Dommermuth in [8] and modified to the two-dimensional scenario by Qing Yang *et al.* in [28, 27, 26] for use in tracking ground-based targets through an acoustic WSN.

Figure 13.8: This figure shows the USB microphone which acts as an acoustic sensor for the PC104 sensor node.

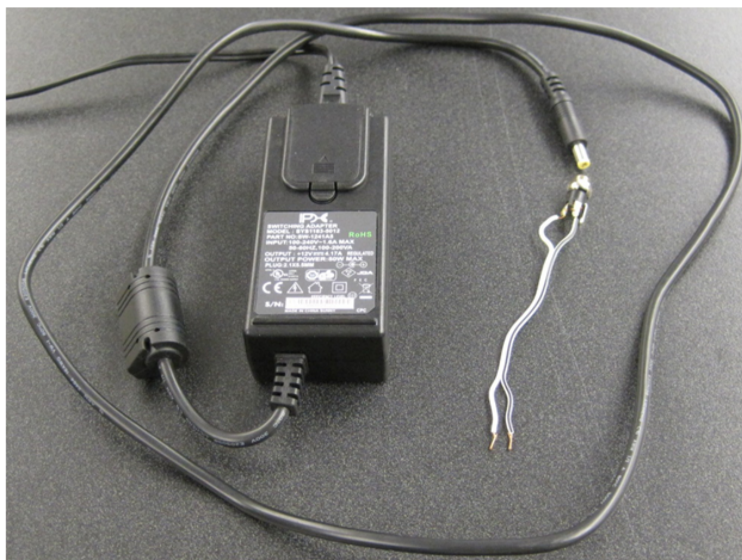


Figure 13.9: This figure shows the +12V AC power supply for the PC104 sensor node used in the lab.

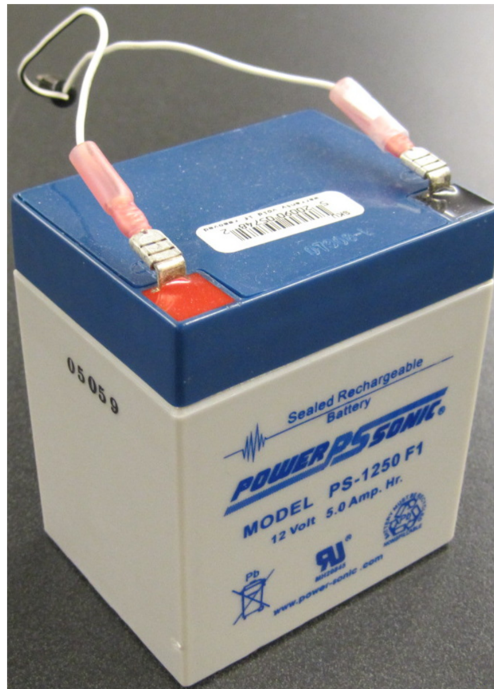


Figure 13.10: This figure shows the +12V battery used to supply power to the PC104 sensor node out in the field.

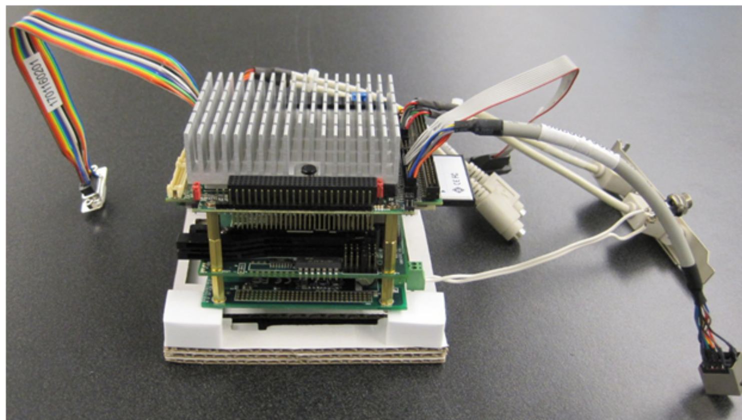


Figure 13.11: This figure shows the completed PC104 sensor node without its protective casing.

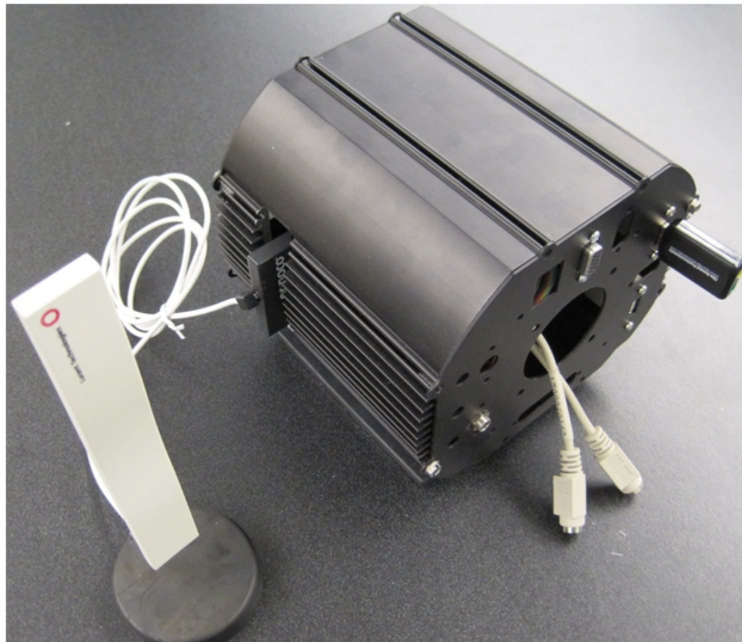


Figure 13.12: This figure shows the completed PC104 sensor node with its protective casing.

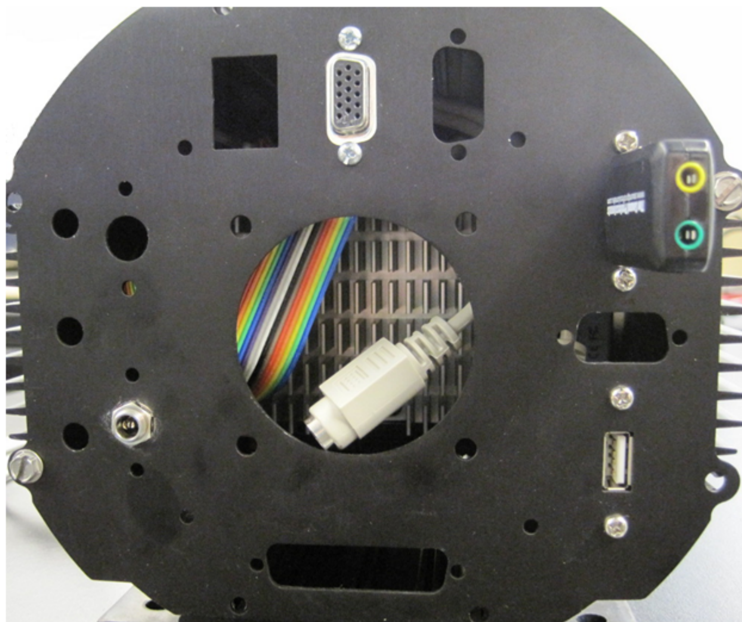


Figure 13.13: This figure shows a close-up of the completed PC104 sensor node in its protective casing.

The sensor network application is easy enough to understand: Each sensor node records acoustic sound signals when a target travels through the acoustic sensors. Each time-synchronized, location-aware sensor node decides the time stamp at which the moving target's sound signal is maximal — presumably the time stamp when the target is physically closest to that acoustic sensor node. This is known as the CPA (closest-point-of-approach) time for that sensor node. This CPA time and sensor location data is relayed from each acoustic sensor node to a cluster head which has the responsibility of actually calculating the target's location (the point at which the moving target is closest to the cluster head), speed and direction.

Figure 13.14 shows a sample wireless sensor network deployment scenario. The lightning bolt shows the wireless communication links between the nodes. Note the "X" in the center of the dotted circle. This position is the returned location — in the form of latitude and longitude — from the cluster head. The cluster head considers the GPS (Global Positioning System) coordinates of latitude and longitude to correspond with an imaginary X-axis and Y-axis, respectively, which spans the entire globe. The direction returned by the cluster head corresponds to the slope of the path of travel made by the moving target. The slope is of the target's path is defined in the usual way of rise divided by run. Or, in this case, the slope is defined to be the change in latitude divided by the change in longitude. The speed, returned by the cluster head in miles per hour, is also calculated using Qing *et al.*'s algorithm.

At this point, only part of the system has been described by explaining the role of the sensor node applications. To complete the description of the entire system, an explanation must now be given of the main External Agent which drives these sensor node applications. An External Agent known as the camera controller is responsible for setting these two sensor node applications into motion by subscribing to "TARG_INFO" via the gateway node. Using the time stamp and the data fields of the named data "TARG_INFO", the camera controller

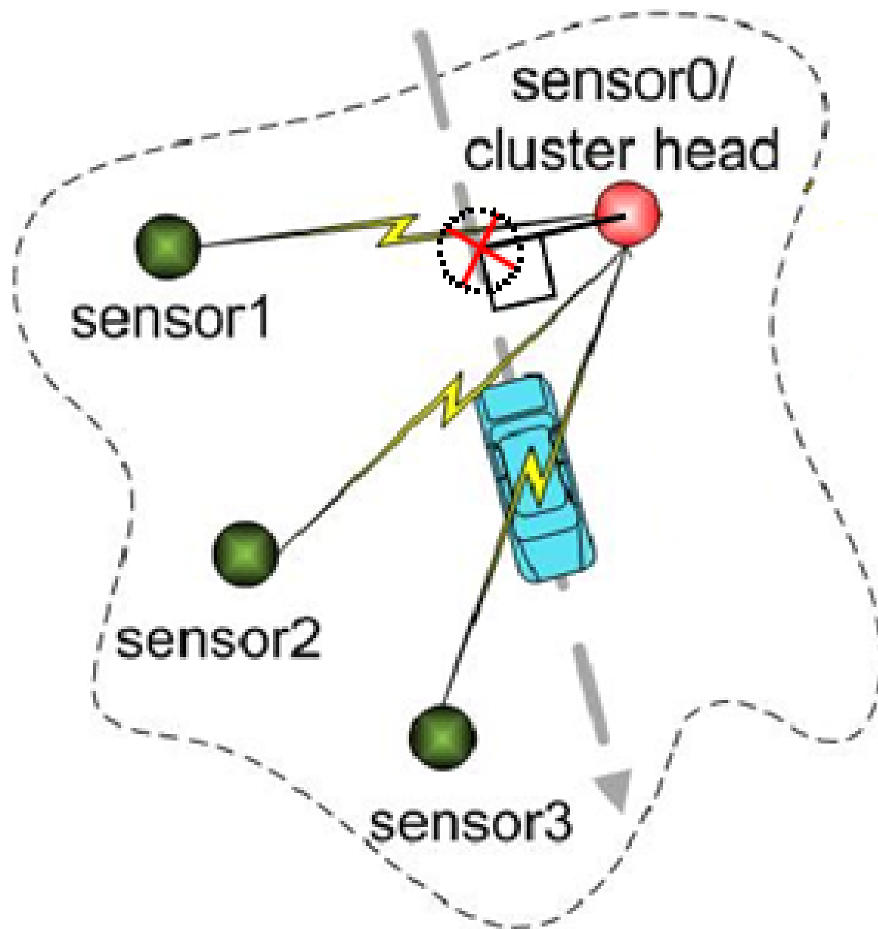


Figure 13.14: This figure shows a sample deployment of a WSN showing the wireless communication links with a lightning bolt. The “X” in the dotted circle shows the location that is calculated by the cluster head. Note that the “X” is directly out from the cluster head at a right angle to the target’s path of travel.

can calculate the current position of the moving target to enable a pan-tilt-zoom camera (shown in Figure 13.15) to swivel to capture video or images of the moving target.

Figure 13.16 shows the entire WSN including the main External Agent, the camera controller. The camera controller and the gateway node is attached to an improvised IP Ethernet network to allow the External Agent API to send subscriptions for the named data “TARG_INFO” to the gateway. A video capture PC (Personal Computer) is connected to the PTZ camera through an RCA-to-USB cable. Because the PC104s have relatively little storage, it is necessary to use this video capture PC to store the large video files of the

Figure 13.15: This figure shows the Sony EVI-D30 Pan-Tilt-Zoom camera controlled by the camera controller External Agent.

moving target captured by the PTZ camera. For our experiments, we assume the video capture unit is always connected to the PTZ camera via the RCA-to-USB cable. When the camera controller software starts up, it first gains access to the PTZ camera via the serial cable. After successfully gaining access to the PTZ camera, the camera controller sends a subscription for the named data "TARG_INFO" to the gateway node.

In the following subsection, a description will be given of the two sensor node applications.

13.2.1 Sensor Node Applications

In this subsection, a description will be given of the two sensor node applications and how they relate. The main code will also be given so the reader can see exactly how one of these real-world applications is written for the actual WSN environment.

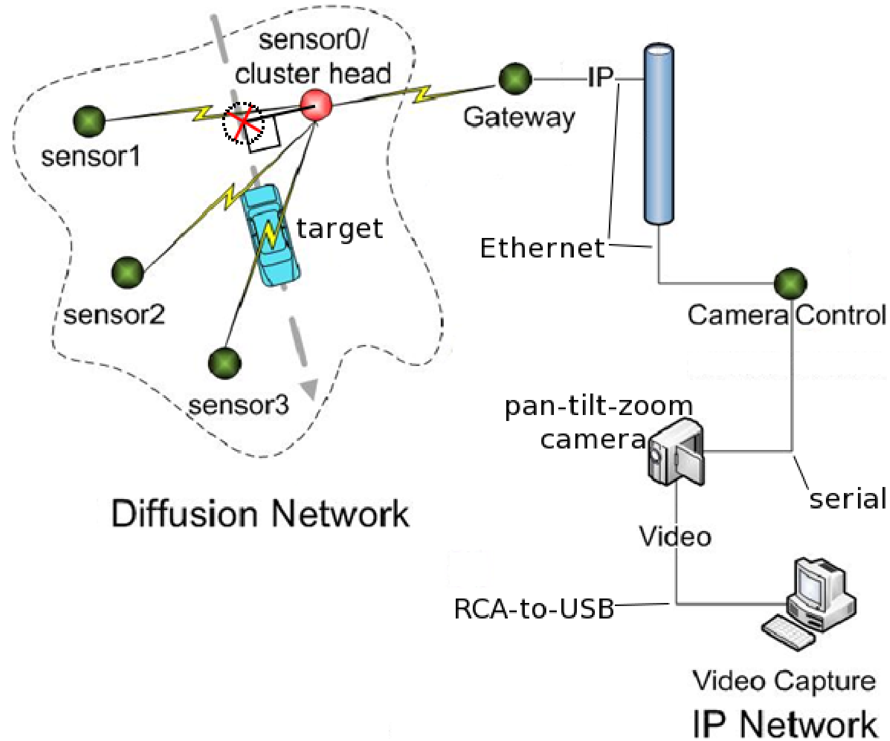


Figure 13.16: This figure shows the experimental setup including the entire network architecture.

"CPA_INFO" Simple Producer

First, there is the sensor node application which produces "CPA_INFO." This application, a simple producer, when tasked, simply waits for a target to appear.

The sensor node application code which produces "CPA_INFO" is shown thus:

```
1: #include "dynServAPI.h"
2: #include "cpadetector.h"
3: Data cpaData;
4: int myMsgQueueID, dsMsgQueueID;
5: char myAppID[MAX_APP_ID_LENGTH];
6: char *toProduce = "CPA_INFO";
7: int main() {
8:   REGISTER_DATA_TO_PRODUCE(toProduce, &myMsgQueueID, myAppID,
```

```
        &dsMsgQueueID);
 9:   while (1) {
10:     cout << "waiting to be tasked" << endl;
11:     AWAIT_TASK(myMsgQueueID);
12:     CPADetector *cpa = new CPADetector();
13:     /* these three values do not change, so just set them once */
14:     strcpy(cpaData.dataType, toProduce);
15:     cpaData.DataUnion.cpaStruct.myLat = getLatitude();
16:     cpaData.DataUnion.cpaStruct.myLong = getLongitude();
17:     bool keepProducing = true;
18:     while (keepProducing == true) {
19:       /* program will block here waiting for a target to appear */
20:       cpa->daemon();
21:       /* if daemon returns, target has appeared and
              CPA time is ready */
22:       cpaData.DataUnion.cpaStruct.cpatime = cpa->getCPATime();
23:       if (CHECK_PUBLISH_DATA_STATUS(dsMsgQueueID, myMsgQueueID,
              myAppID, toProduce) == -1) {
24:         keepProducing = false;
25:         delete cpa;
26:       } else {
27:         /* CONTINUE producing data" */
28:         PUBLISH_DATA(dsMsgQueueID, &cpaData);
29:       }
30:     }
31:   }
32:   return EXIT_SUCCESS;
```

```
33: }
```

This simple producer is tasked to produce "CPA_INFO." This can be understood through taking a look at Figure 13.17. This figure shows a graph of the sound intensity as a target approaches the sensor node, moves through the closest-point-of-approach (CPA) relative to the sensor node and begins moving away from the node. The vertical line represents the time at which the target is at its closest-point-of-approach. This time stamp is that which is produced by this sensor application and is forwarded to the cluster head.

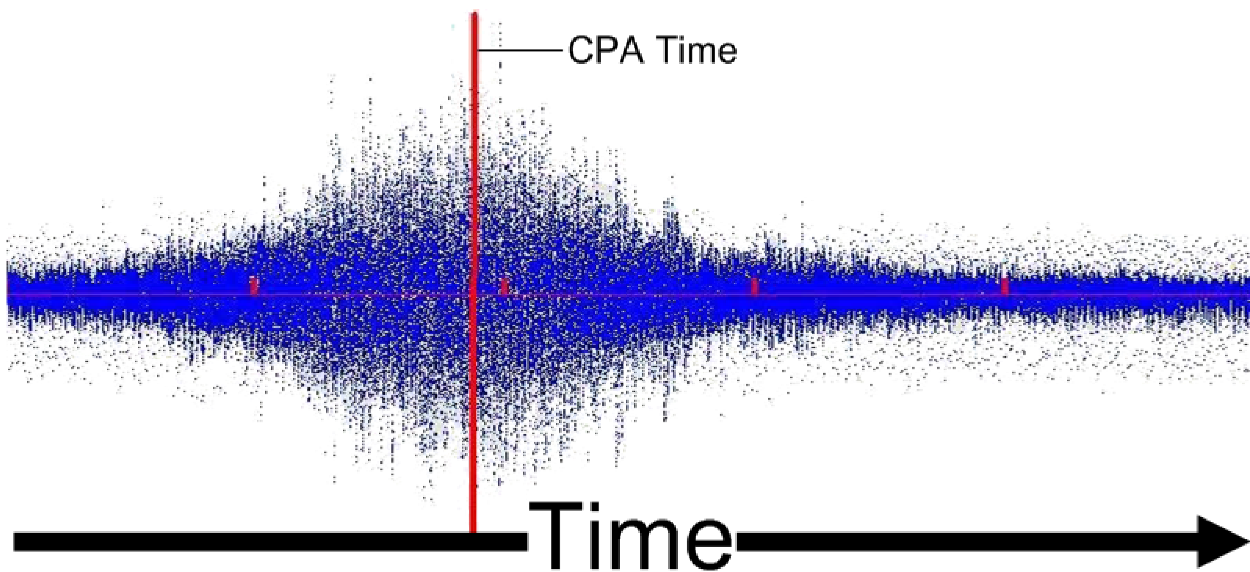


Figure 13.17: This figure shows the sound signature and closest-point-of-approach time of a target moving relative to a single acoustic sensor.

"TARG_INFO" Complex Producer

Next, there is the sensor node application which actually calculates the target's location, speed and direction using "CPA_INFO" obtained from other sensor nodes. This sensor node is known as the cluster head and produces the named data "TARG_INFO." For the cluster head, a complex producer, to produce the named data "TARG_INFO," it must have the

named data "CPA_INFO" to perform its job. Therefore, once the cluster head is tasked, it must subscribe to "CPA_INFO" in order to produce "TARG_INFO."

The sensor node application which depends on the named data "CPA_INFO" to calculate the target's location, speed and direction to produce its named data "TARG_INFO" is shown thus:

```
 1: #include "dynServAPI.h"
 2: #include "targInfoApp.h"
 3: int myMsgQueueID, dsMsgQueueID;
 4: char myAppID[MAX_APP_ID_LENGTH];
 5: char *toProduce = "TARG_INFO";
 6: char *toConsume = "CPA_INFO";
 7: int main() {
 8:   REGISTER_DATA_TO_PRODUCE(toProduce, &myMsgQueueID, myAppID,
        &dsMsgQueueID);
 9:   while (1) {
10:     cout << "waiting to be tasked" << endl;
11:     AWAIT_TASK(myMsgQueueID);
12:     vector<string> vecOfSubs;
13:     vecOfSubs.push_back(string(toConsume));
14:     SUBSCRIBE_TO_DATA(&vecOfSubs, dsMsgQueueID, myAppID);
15:     targInfoApp *ta = new targInfoApp();
16:     Data cpaInfoData;
17:     bool keepProducing = true;
18:     while (keepProducing == true) {
19:       AWAIT_DATA(myMsgQueueID, &cpaInfoData);
20:       if (CHECK_PUBLISH_DATA_STATUS(dsMsgQueueID, myMsgQueueID,
              myAppID, toProduce) == -1) {
```

```
21:         keepProducing = false;
22:         delete ta;
23:         UNSUBSCRIBE_FROM_DATA(&vecOfSubs, dsMsgQueueID,
              myAppID);
24:       } else {
25:         /* CONTINUE producing data */
26:         ta->recvCPAData(&cpaInfoData);
27:       }
28:     }
29:   }
30:   return EXIT_SUCCESS;
31: }
32: void targInfoApp::recvCPAData(Data *data) {
33:   /* perform calculations and store them in class variables */
34:   if (/*received sufficient "CPA_INFO" from different nodes*/) {
35:     Data targInfoData;
36:     strcpy(targInfoData.dataType, toProduce);
37:     targInfoData.DataUnion.targInfoStruct.Lat = this.getLat();
38:     targInfoData.DataUnion.targInfoStruct.Long = this.getLong();
39:     targInfoData.DataUnion.targInfoStruct.Speed = this.getSpeed();
40:     targInfoData.DataUnion.targInfoStruct.Direction =
          this.getDirection();
41:     targInfoData.DataUnion.targInfoStruct.CPATime =
          this.getCPATime();
42:     PUBLISH_DATA(dsMsgQueueID, &visualData);
43:   }
44: }
```

This concludes the details pertaining to the sensor node applications located inside the WSN. Next, a discussion of the camera controller application and introduction of an application for logging and monitoring will be given.

13.2.2 External Agent Applications

Target Tracking

An External Agent application which relies on the named data "TARG_INFO" has also been implemented. In a nutshell, the External Agent application uses the named data "TARG_INFO" from a WSN to drive a PTZ camera to track a target moving. The main camera controller code to track a moving target is given thus:

```
 1: #include "EVI-D30.h"
 2: #include "externalAgentAPI.h"
 3: void SendInfoToCamera(long Lat, long Long, double Speed, double Slope,
      double timeStamp);
 4: int socketDesc;
 5: vector<struct sockaddr_in> remoteGatewaysVec;
 6: char *toConsume = "TARG_INFO";
 7: /* allows access to Sony EVI-D30 PTZ camera */
 8: EVI_D30 *cam;
 9: int main() {
10:   /* gain access to the camera */
11:   cam = NULL;
12:   cam = new EVI_D30();
13:   if (cam == NULL) {
14:     DieWithError("cannot create internal camera structure");
15:   }
16:   /* open serial port */
```

```
17:   if (cam->Init() != 1) {
18:     DieWithError("cannot initialize camera structure");
19:   }
20:   char serialPort[20] = "/dev/ttyS0";
21:   if (cam->Open(1, serialPort) != 1) {
22:     DieWithError("cannot access to camera through serial port");
23:   }
24:   /* grab the information about the camera's orientation
        and position */
25:   long camLat = getLat();
26:   long camLong = getLong();
27:   string camOrientation = getOrientation();
28:   /* set up remote WSN gateway info */
29:   struct sockaddr_in gatewayAddr;
30:   memset(&gatewayAddr, 0, sizeof(gatewayAddr));
31:   gatewayAddr.sin_family = AF_INET;
32:   gatewayAddr.sin_addr.s_addr = inet_addr("131.204.142.200");
33:   gatewayAddr.sin_port = htons(8989);
34:   remoteGatewaysVec.push_back(gatewayAddr);
35:   /* set up subscription to gateway */
36:   vector<string> dataTypeVec;
37:   dataTypeVec.push_back(string(toConsume));
38:   if ((socketDesc = GATEWAY_SUBSCRIBE(&remoteGatewaysVec,
          &dataTypeVec)) < 0) {
39:     DieWithError("GATEWAY_SUBSCRIBE failed");
40:   }
41:   Data pktFromGateway;
```

```
42:   while (1) {
43:     cout << "awaiting data from gateway" << endl;
44:     if (RECEIVE_FROM_GATEWAY(socketDesc, &pktFromGateway) < 0) {
45:       DieWithError("RECEIVE_FROM_GATEWAY failed");
46:     }
47:     if (strcmp(pktFromGateway.dataType, toConsume) == 0) {
48:       double ts = pktFromGateway.DataUnion.TARG_INFOStruct.CPATime;
49:       long Lat = pktFromGateway.DataUnion.TARG_INFOStruct.Lat;
50:       long Long = pktFromGateway.DataUnion.TARG_INFOStruct.Long;
51:       double Speed = pktFromGateway.DataUnion.TARG_INFOStruct.Speed;
52:       double Slope = pktFromGateway.DataUnion.TARG_INFOStruct.Slope;
53:       SendInfoToCamera(Lat, Long, Speed, Slope, ts);
54:     } else {
55:       /* drop unexpected data type */
56:     }
57:   }
58:   return 1;
59: }
```

Logging and Monitoring

The logging and monitoring application implemented simply subscribes to all of the data types the WSN can produce. Upon receiving named data, the application checks the named data type and simply writes its contents to an appropriate log file for long-term storage or further offline analysis at a later time. The application code is given thus:

```
 1: #include "externalAgentAPI.h"
 2: int socketDesc;
 3: char *targInfo = "TARG_INFO";
```

```
 4: char *cpaInfo = "CPA_INFO";
 5: int main() {
 6:   /* set up log files */
 7:   ofstream targInfoLog("./TARG_INFO_log.txt", ios_base::app);
 8:   ofstream cpaInfoLog("./CPA_INFO_log.txt", ios_base::app);
 9:   /* set up remote WSN gateway info */
10:   vector<struct sockaddr_in> remoteGatewaysVec;
11:   struct sockaddr_in gatewayAddr;
12:   memset(&gatewayAddr, 0, sizeof(gatewayAddr));
13:   gatewayAddr.sin_family = AF_INET;
14:   gatewayAddr.sin_addr.s_addr = inet_addr("131.204.142.200");
15:   gatewayAddr.sin_port = htons(8989);
16:   remoteGatewaysVec.push_back(gatewayAddr);
17:   /* setup subscriptions to gateway */
18:   vector<string> subVec;
19:   subVec.push_back(string(targInfo));
20:   subVec.push_back(string(cpaInfo));
21:   if ((socketDesc = GATEWAY_SUBSCRIBE(&remoteGatewaysVec,
          &subVec)) < 0) {
22:     DieWithError("GATEWAY_SUBSCRIBE failed");
23:   }
24:   Data dataFromWSN;
25:   time_t ts;
26:   while (1/* or some other halting criteria */) {
27:     cout << "Waiting for data ..." << endl;
28:     if (RECEIVE_FROM_GATEWAY(socketDesc, &dataFromWSN) < 0) {
29:       DieWithError("RECEIVE_FROM_GATEWAY failed");
```

```
30:     }
31:     time(&ts);
32:     if (strcmp(dataFromWSN.dataType, targInfo) == 0) {
33:       printTargInfoToFile(targInfoLog, ts, &dataFromWSN);
34:     } else if (strcmp(dataFromWSN.dataType, cpaInfo) == 0) {
35:       printCpaInfoToFile(cpaInfoLog, ts, &dataFromWSN);
36:     } else {
37:       /* drop unrecognized data types */
38:     }
39:   }
40:   return 1;
41: }
```

13.3 Target Tracking Experimental Results

Table 13.1 and Figure 13.18 show experimental results of several experiments where a moving target travels through the acoustic WSN used for target tracking.

Table 13.1 shows the results of eight experiment runs with the error for position, speed and angle of the target. The minimum, maximum and average error for each of target position, speed and angle is aggregated at the bottom of the table.

Figure 13.18 shows the expected and actual lines of travel of the moving target. As you can see, the WSN can quite accurately calculate the target's CPA position, speed and direction with very little error. According to the authors of [29], Differential Global Positioning System (GPS) is only accurate to around three feet on average. According to Table 13.1, our target tracking system based on gathering CPA data from acoustic sensors is nearly as accurate as the GPS system.

Figure 13.19 shows a snapshot taken of a moving target after it has been detected in the acoustic WSN. After the named data "TARG_INFO" reaches the camera controller External

Run	Target Position (feet) Computed	Error	Speed (mph) Computed	Error	Angle (degree) Computed	Error
1st	(3608558, 641360)	3.3	28	2	90	3
2nd	(3608558, 641360)	3.3	27	3	90	3
3rd	(3608559, 641357)	7.3	42	12	84	3
4th	(3608552, 641359)	19.7	27	3	111	24
5th	(3608545, 641358)	3.3	29	1	127	1
6th	(3608544, 641359)	3.3	28	2	130	2
7th	(3608544, 641359)	3.3	29	1	131	3
8th	(3608545, 641360)	3.3	32	2	132	4
Min		**3.3**		**1**		**1**
Max		**19.7**		**12**		**24**
Avg		**5.85**		**3.25**		**5.375**

Table 13.1: Target Tracking Experimental Results
This table shows the test results of several runs of a target through the WSN dedicated to tracking moving targets.

Agent, snapshots are taken of the target. This is one such snapshot. In another application, video could be taken and recorded or relayed to a manned base station for further analysis or action.

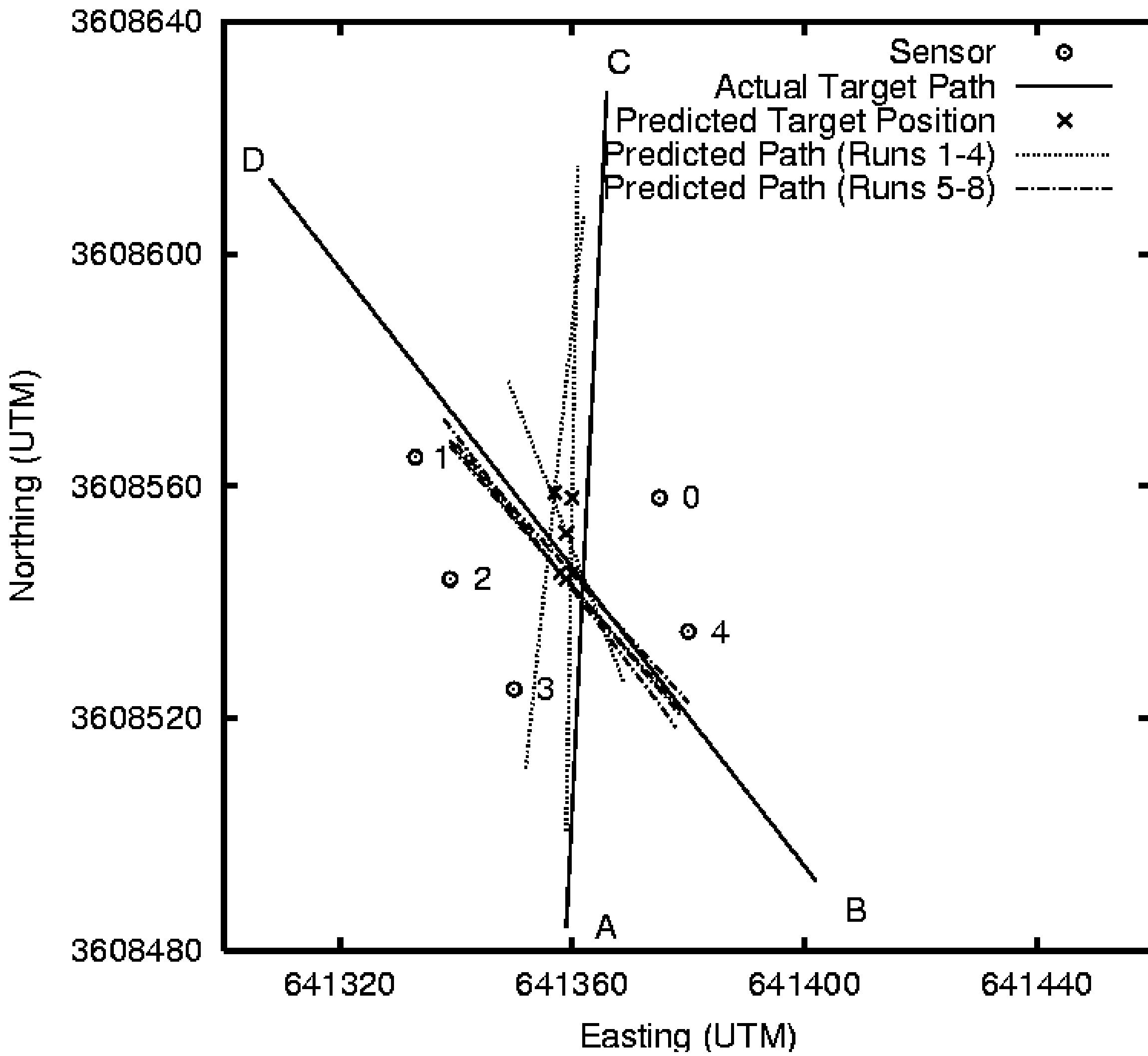


Figure 13.18: This figure shows the expected and actual lines of travel of the moving target in the target tracking experiment. Please refer to Table 13.1 for the target attributes and errors resulting from experiment runs 1–4 and 5–8 shown here.

Figure 13.19: This figure shows a snapshot of a moving target after it has been detected by the acoustic WSN. In this frame, the camera has rotated in order to have the target's position within its viewing range.

Chapter 14

Experiment Design & Performance Evaluation

In this chapter details of the experiment design and performance evaluation are given.

14.1 Experiment Design

To evaluate the performance of DS, it is necessary to evaluate the impact of DS on sensor node applications. In particular, assurance must be made that the DS service layer does not impact the time to task a sensor application nor the time for a sensor application to publish data.

A relatively simple timing analysis is employed to evaluate the impact of DS on sensor node applications' ability to be tasked and to publish data. Recall that there are two threads running in any pure DD application: a tasking thread and a main thread. It is critical to take time stamps for the timing analysis at appropriate points inside these threads.

First, a consideration is made for the *tasking time* of pure DD and DS sensor node applications. For pure DD sensor applications, the tasking time is defined as the time between when the tasking thread is first entered and when the main thread realizes it has been tasked. For DS sensor applications, tasking time is defined as the time between when DS's tasking thread is first entered and when the application realizes it has been tasked. For DS, this will give us an idea of the length of time it takes for this information to travel through DS, through the Message Queue and into the sensor application.

Publishing time is defined as the time between when the sensor node application is tasked and when the data is handed over to DD for network transmission. For pure DD sensor applications, the publishing time is the time between when the application is tasked and when the application finished handing over the data to DD. For DS sensor applications,

the publishing time is the time between when the application is tasked and when DS hands over the data to DD. For DS, this will give us an idea of the length of time it takes for this information to travel through DS's Message Queue and through DS to the DD network.

14.2 Performance Evaluation

From the beginning, one of the goals has been to reduce the workload of DD application programmers by simplifying their programming task. With the DS API, there is no need for sensor node application programmers to understand the complexities of the pure DD API. Before the timing analysis is presented, let us first compare the complexity of sensor application code which uses the DS API versus sensor application code which uses the pure DD API.

In the following subsections a description is given for what is meant by the claim that the DS API makes programming over this type of system easier and give some very simple performance metrics which show that the overhead incurred by DS and the DS API is negligible when compared to that of using the pure DD API.

It should be said that it does not make sense to compare External Agent code "before" and "after" DS because, before DS, there is no concept of an External Agent. Therefore, it is only possible to compare a sensor application's LOC count before and after DS was implemented. This thesis takes on the LOC metric in determining to what degree programming using the DS API is more or less `easy` compared to not using the DS API.

14.2.1 LOC Metric

The LOC count comparison is given in Table 14.1 for two different sensor node applications, a sensor node application which produces the named data "CPA_INFO" and the other application which depends on "CPA_DATA" to produce its named data "TARG_INFO."

For the sensor node application which produces the named data "CPA_INFO," the number of LOC required just to complete the implementation part which directly uses the

Sensor Node Application Name	**LOC *Without* DS**	**LOC *With* DS**
"CPA_INFO" Application	>50	<10
"TARG_INFO" Application	>65	<15

Table 14.1: This table compares the lines of code required to write two different applications both with and without the DS API. Note that application-specific code is not counted.

DD API was more than 50 lines of code. Compare 50 LOC with less than 10 LOC, the number of LOC to write the same sensor application using the DS API. For the sensor node application which produces the named data "TARG_INFO," the number of LOC to complete the sensor node implementation using the pure DD API was more than 65 lines of code. With DS, this number is reduced to less than 15 LOC. Using LOC count as the metric to determine the relative "easiness" of building sensor node applications, it is clear that the new DS API simplifies the programming task of sensor node applications. These data are summarized in Table 14.1.

14.2.2 DS Delay Analysis

Another metric of interest is how much overhead is incurred by the DS service layer. For this investigation, an application was written both *with* and *without* DS to investigate the relative delay of first being tasked and publishing the first data packet onto the DD network.

Let us first see the definitions of *tasking time* and *publishing time* as they are defined with using DS with two figures (Figures 14.1 and 14.2, respectively). *Tasking time* with the DS service layer (Figure 14.1) is defined as the length of time between when Dynamic Services' tasking thread is first invoked after DD receives an interest from the network and when the sensor node application is aware there is some interest in the data it can produce. *Publishing time* with the DS service layer (Figure 14.2) is defined as the length of time between when the application is first tasked and when the publish success of the first published data packet is returned from DS.

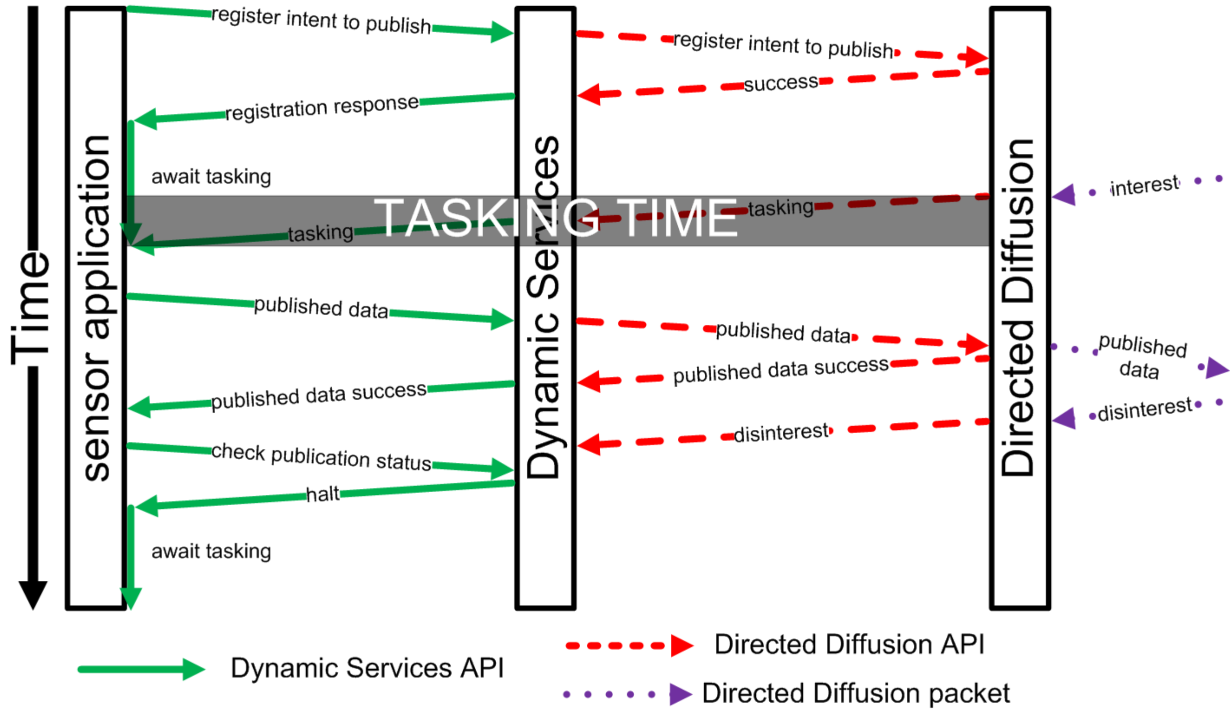


Figure 14.1: This figure shows *tasking time* for sensor node applications with Dynamic Services. Tasking time with DS is defined as the length of time between when Dynamic Services' tasking thread is first invoked after DD receives an interest from the network and when the sensor node application is aware there is some interest in the data it can produce.

Next, let us see the definitions of *tasking time* and *publishing time* as they are defined without using DS with two figures (Figures 14.3 and 14.4, respectively). *Tasking time* without the DS service layer (Figure 14.3) is defined as the length of time between when a sensor application's tasking thread is invoked after an interest is received from the DD network and when the application's main thread realizes its shared variable has been updated. *Publishing time* without the DS service layer (Figure 14.4) is defined as the length of time between when the application's main thread realizes its shared variable has been updated and when the publish success of the first published data packet is returned from DD.

Figures 14.5 and 14.6 show the tasking time and publishing time, respectively, incurred by the sensor application both with and without DS. Figure 14.5 shows the tasking time (in milliseconds) incurred. In the case of tasking, DS has the ability to task the application in less than one millisecond whereas the busy waiting, or polling, of pure DD applications

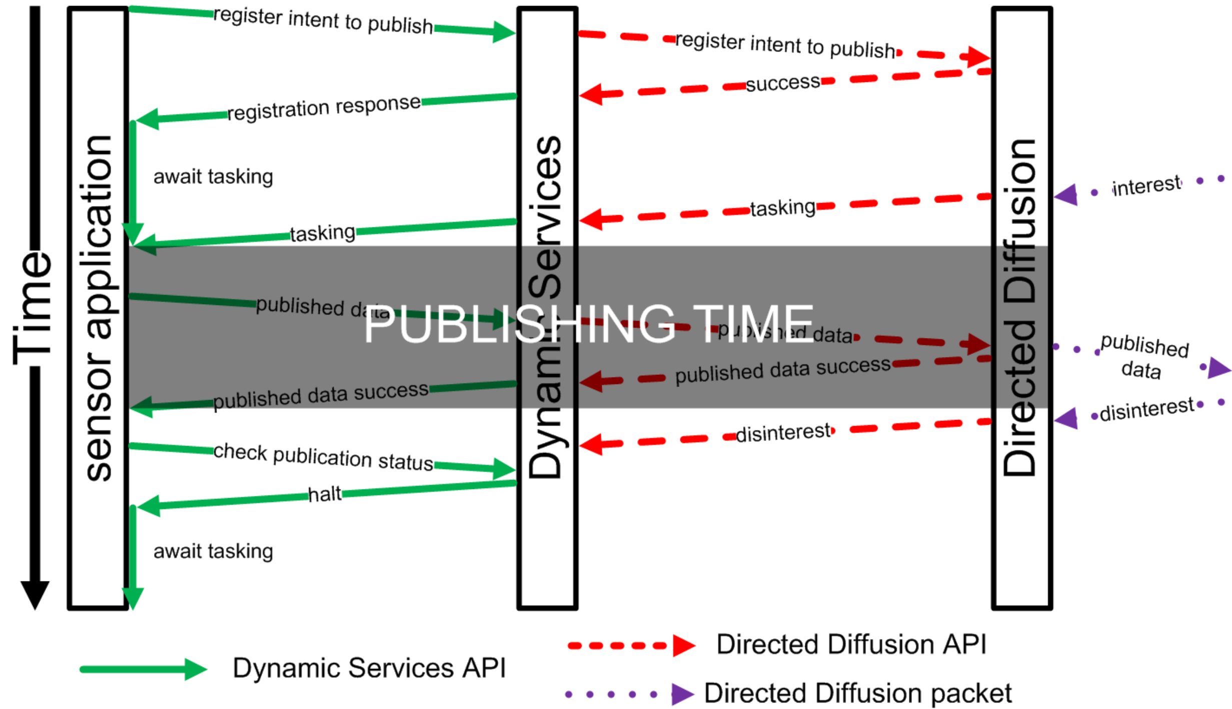


Figure 14.2: This figure shows *publishing time* for sensor node applications with Dynamic Services. Publishing time with DS is defined as the length of time between when the application is first tasked and when the publish success of the first data packet is returned from DS.

increases the time to task the application dramatically as the number of running sensor applications increases.

It is important to note here that the internal clock on the PC104 sensor nodes on which these experiments were carried out were accurate only to the millisecond. On these graphs, a plotted point of zero milliseconds merely means *less than* one millisecond.

Figure 14.6 shows the delay incurred in the publishing process. With only a few applications running, DS publishes just as quickly as pure DD applications. As the number of running applications increases, however, it becomes clear that DS is the winner. Since DS sensor applications are sleeping while awaiting tasking at their own Message Queues rather than busily waiting like pure DD sensor applications, the publishing time remains consistently better than the publishing time for pure DD sensor applications because it is not necessary for the tasked sensor node application to compete for CPU cycles.

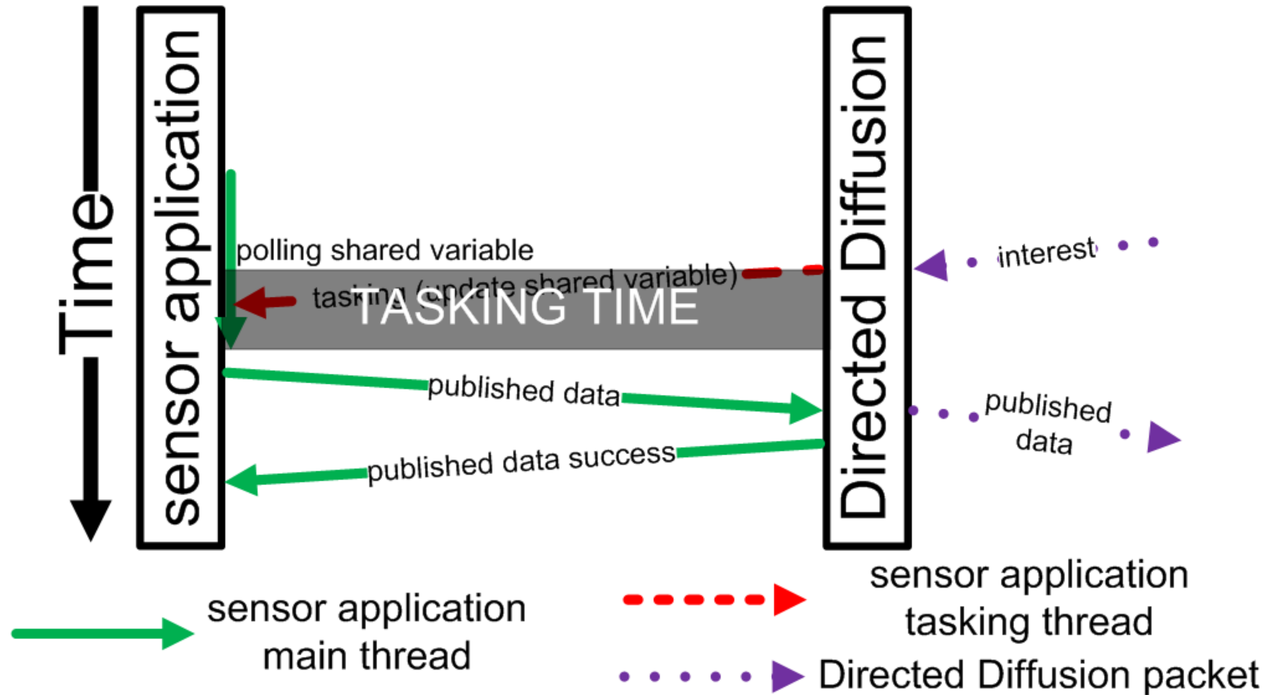


Figure 14.3: This figure shows *tasking time* for sensor node applications without Dynamic Services. Tasking time without DS is defined as the length of time between when sensor application's tasking thread is invoked after an interest is received from the DD network and when the application's main thread realizes its shared variable has been updated.

Just by looking at these data graphs — where a lower plotted value is better in terms of delay — it is clear that the Message Queues provided by the Linux kernel of the Slax OS used in DS and the DS API actually make the operations of tasking and publishing quicker. Rather than the Slax CPU scheduler scheduling processes to busily wait and waste CPU and energy resources like in pure DD sensor applications, DS sensor applications sleep awaiting for tasking at Message Queues. This is why it actually takes pure DD applications longer to be tasked and to publish. This is a nice feature of Message Queues in that they eliminate the need for busy waiting.

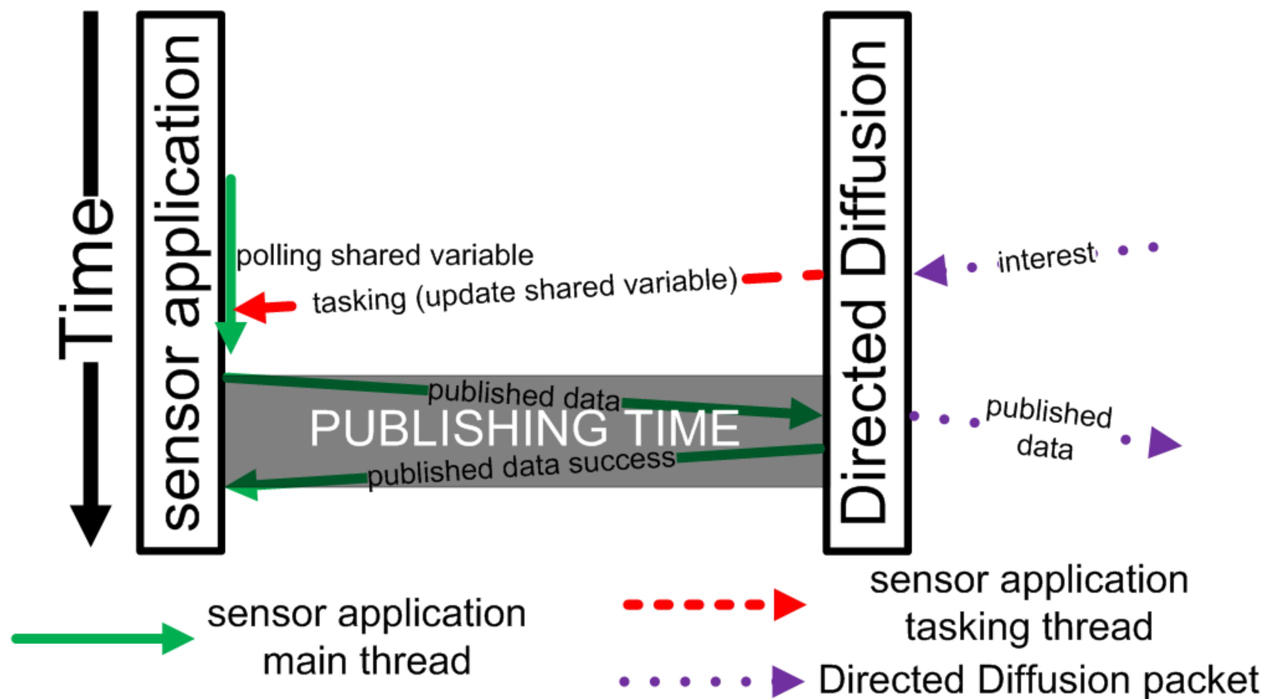


Figure 14.4: This figure shows *publishing time* for sensor node applications without Dynamic Services. Publishing time without DS is defined as the length of time between when the application's main thread realizes its shared variable has been updated and when the publish success of the first data packet is returned from DD.

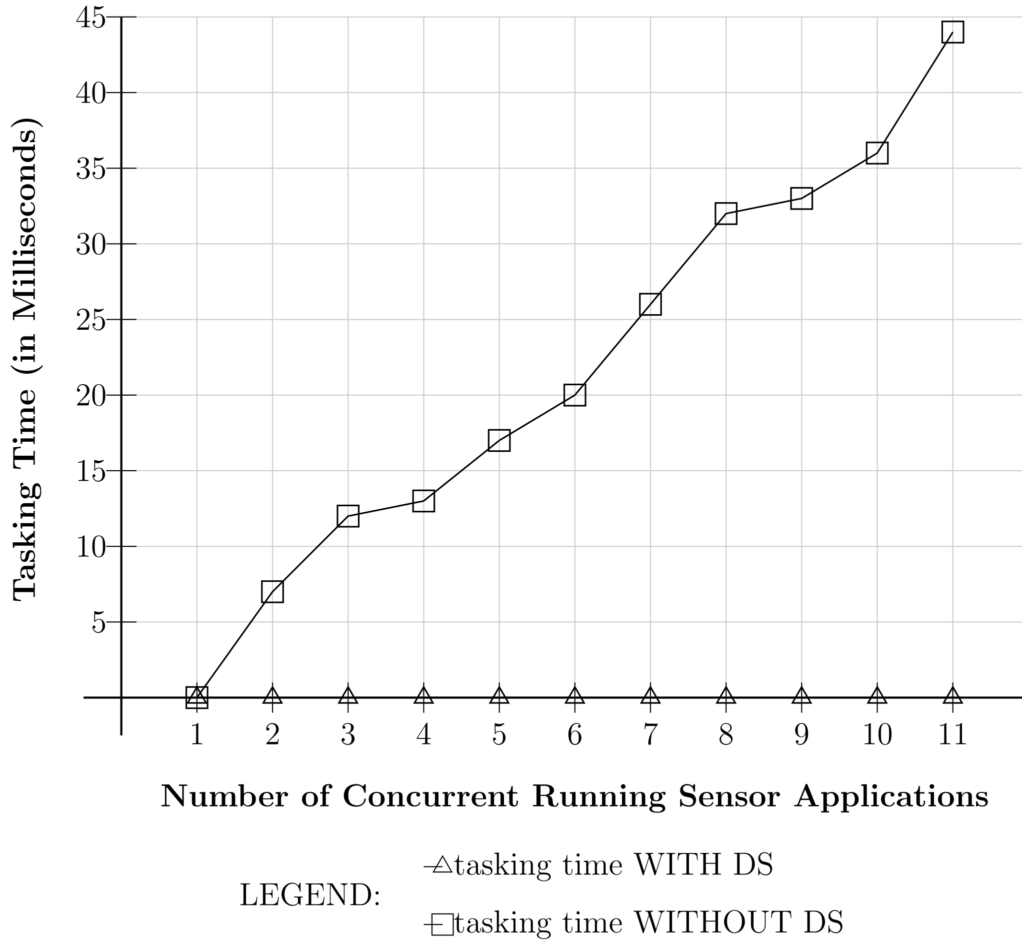


Figure 14.5: This graph shows the tasking time for sensor applications both with and without Dynamic Services as the number of running sensor applications varies. Since DD sensor applications busily wait, the tasking time of pure DD sensor applications is greater. A lower plotted point on the graph is better in terms of delay than a higher plotted point.

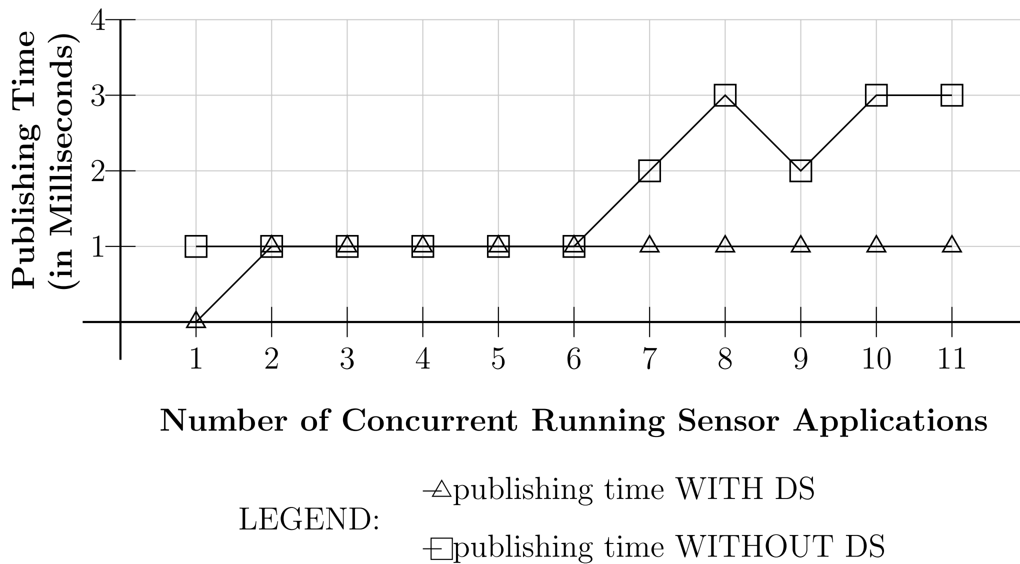


Figure 14.6: This graph shows the publishing time for sensor applications both with and without Dynamic Services as the number of running sensor applications varies. The publishing time remains consistent for DS sensor applications as the number of sensor applications increases whereas for pure DD sensor applications, busy waiting takes a toll on performance. A lower plotted point on the graph is better in terms of delay than a higher plotted point.

Chapter 15

Future Work

Future work could follow many different paths. As mentioned earlier in this thesis, there are many different roles at play within the entire system; therefore, a few avenues of future work is given for sensor node applications, the application-layer gateway node and External Agents individually.

15.1 Concerning Sensor Applications

Further research could be done on extending the DS API to include all of the flexibility of the pure DD API. One could also port DS to other data-centric networking protocols or come up with other schemes which enables DS to be used over non-data-centric networking protocols.

Further work could be done in allowing sensor nodes in a WSN to submit interests for named data types to *other* remote WSNs. For this scheme to work, gateway nodes would have to be aware of one another to facilitate sending of interests and data to and from remote WSNs. This would inherently create complexity in the gateway nodes, but sensor node applications would largely be unaware of this complexity. Sensor applications would simply do what they have always done.

Further work could also be done in allowing sensor applications to submit interests for named data to External Agents. This scheme would require External Agents to register with gateway nodes and would create more complexity in the External Agents. This could be prohibitive for smaller, hand-held devices.

Currently, the DS API does not support guaranteed delivery of data. DD is a best effort service but essentially does not guarantee delivery of data. The DS API could be extended to ensure guaranteed delivery of data to the gateway node or to other sensor nodes.

15.2 Concerning the Application-Layer Gateway

As was previously mentioned, the gateway-based solution provided in this thesis is an application-layer gateway. Work could be done on making the application-layer gateway implementation be a network-layer gateway. This would require mapping an IP address in an IP packet to a named data type at the network-layer gateway. Currently, there is no technology available which implements any such mapping. One idea is to have sensor applications register with the network-layer gateway node so the gateway node can assign specific IP addresses to certain named data types. External agents could request the names of named data types from this network-layer gateway node and the corresponding IP addresses of these data types would be sent back to the External Agent. An External Agent could use this information to request data types by IP address only.

15.3 Concerning External Agent Applications

The current External Agent API only supports UDP. Extensions could be made to the API that also allow for TCP connections between the External Agent and the remote gateway node. The External Agent API could also be extended in such a way as to allows External Agents to request guaranteed delivery within the WSN using DD's RMST, for example. Combining DD's RMST with a TCP connection between the External Agent and the gateway node could guarantee delivery of packets from individual sensor node applications to External Agents who desire this service.

Chapter 16

Conclusions

This thesis has presented an architecture which allows IP-based hosts to easily task and harvest data from remote Dynamic Services-enabled Directed Diffusion wireless sensor networks. A presentation of APIs for both IP-based and sensor node application programmers has been given. The DS middleware service layer which eases sensor node application programmers' programming tasks and a performance analysis of the system showing the relative benefit of using the DS service layer has also been given. Results of an "ease of programming" metric to show the relative benefit to programmers when using the DS API versus programming directly over the DD protocol was also given.

A description of an application-layer gateway has been given which is used to enable IP-based hosts to gather data from one or more remote WSNs. Through using an IP address-to-named data type map inside of the gateway node, IP-based hosts can gather data from the data-centric DD wireless sensor network through the use of a simple API.

A LOC count comparison has been given comparing the pure DD API versus using the DS API presented in this thesis. In the two applications implemented using both the pure DD API and the DS API, it has been shown that the DS API significantly reduces the amount of programming work which must be done in implementing a sensor node application versus using the pure DD API. A performance analysis which clearly shows the value of using DS rather than relying on the polling of pure DD sensor applications is also shown. In short, the performance impact of DS, since it mostly relies on System V Message Queues for tasking and publishing, is negligible when compared to the performance of pure DD sensor applications for the same operations and actually improves performance due to the fact that pure DD applications busily wait to be tasked.

All in all, it has been determined that with a flexible API, many more tasks can be accomplished with more complexity while a simple API may cause simple tasks to become more difficult but may allow more devices to use it. There should be a balance struck between these two competing ideals. On one hand, if we make the API too large and complex, it may squeeze out smaller, hand-held devices. However, if we make the API too simple and trivial, we lose the power that comes with greater flexibility.

We have also seen that it is possible to, at least partially, bridge the gap between data-centric networks and host-centric networks like IP. Through using the gateway node's mapping service, it is possible to transfer data from a data-centric WSN to interested IP-based hosts.

Bibliography

[1] A. Dunkels, T. Voigt, N. Bergman, and M. Jnsson. "An IP-based Sensor Network as a Rapidly Deployable Building Security System." In *Swedish National Computer Networking Workshop*, Karlstad, Sweden, November 2004.

[2] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. "Wireless sensor networks for habitat monitoring." In *First ACM Workshop on Wireless Sensor Networks and Applications* (WSNA 2002), Atlanta, GA, USA, September 2002.

[3] Adam Dunkels, Thiemo Voigt, Juan Alonso, Hartmut Ritter, and Jochen Schiller. "Connecting Wireless Sensornets with TCP/IP Networks." In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications* (WWIC2004), Frankfurt (Oder), Germany, February 2004.

[4] Archana Bharathidasan, Vijay Anand Sai Ponduru. "Sensor Networks: An Overview." IEEE Infocom, 2004.

[5] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin. "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks." In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking* (MobiCOM '00), Boston, Mass., August 2000.

[6] CUI Yanrong and CAO Jiaheng. "Power Efficient Data Query Processing Protocol for Wireless Sensor Networks." IEEE, 2007.

[7] D. Culler, D. Estrin, and M. Srivastava, "Overview of sensor networks," *IEEE Computer*, pp. 41–49, August 2004.

[8] F. M. Dommermuth, "The estimation of target motion parameters from cpa time measurements in a field of acoustic sensors," *The Journal of the Acoustical Society of America*, vol. 83, no. 4, pp. 1476–1480, 1988.

[9] Fred Stann and John Heidemann. "RMST: Reliable Data Transport in Sensor Networks," Appearing in *1st IEEE International Worlshop on Sensor Net Protocols and Applications* (SNPA), Anchorage, Alaska, USA. May 11, 2003.

[10] G. J. Pottie and W. J. Kaiser. "Embedding the Internet: Wireless Integrated Network Sensors." *Communications of the ACM*, 43(5):5158, May 2000.

[11] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. "Monitoring volcanic eruptions with a wireless sensor network." In *Proceedings of the Second European Workshop on Sensor Networks*, 2005.

[12] Heinzelman W, Chandrakasan A and Balakrishnan H. "Energy-Efficient Communication Protocol for Wireless Microsensor Networks." In IEEE *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, 2000.

[13] Hui Dai, Richard Han. "Unifying Micro Sensor Networks with the Internet via Overlay Networking," In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (LCN), Tampa, FL, November 2004.

[14] Jin Zhang, Daxin Liu, Yuezhu Xu and Tong Wang. "Dissymmetrical Wireless Sensor Network Internet Access Technology Based on Optimum Gateway." In IEEE *International Conference on Computer Science and Information Technology*, 2008.

[15] Joanna Kulik, Wendi Rabiner and Hari Balakrishnan. "Adaptive Protocols for Information Dissemination in Wireless Sensor Networks." In *MobiCom*, 1999.

[16] Joseph M. Hellerstein, Wei Hong, Samuel Madden and Kyle Stanek. "Beyond Average: Toward Sophisticated Sensing with Queries." In *IPSN*, 2003.

[17] Karim A. Emara, Mohammad Abdeen, Mohammad Hashem. "A Gateway-based Framework for Transparent Interconnection between WSN and IP Network." IEEE, 2009.

[18] Karl Aberer, Manfred Hauswirth, Ali Salehi. "A Middleware for Fast and Flexible Sensor Network Deployment." ACM *VLDB '06*, September 2006.

[19] Samuel Madden, Michael J. Franklin, Joseph Hellerstein and Wei Hong. "TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks." SIGOPS Oper Syst Rev 36(SI):131–146.

[20] Man-Hon Chan, King-Shan Lui and Vincent Tam. "Efficient Event and Query Distribution in Sensor Networks." IEEE, 2005.

[21] Min Zhang, Sangheon Pack, Kideok Cho, Dukhyun Chang, Yanghee Choi, and Taekyoung Kwon, "An Extensible Interworking Architecture (EIA) for Wireless Sensor Networks and Internet," In *Proc. Asia-Pacific Network Operations and Management Symposium* (APNOMS) 2006 Poster Sessions, Busan, Korea, September 2006.

[22] N. Trigoni, Y. Yao, A Demers, J. Gehrke, R. Rajaraman. "Multi-query optimization for sensor networks."

[23] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss and Philip Levis. "Collection Tree Protocol." Technical Report SING-09-01.

[24] Pardeep K. Mohanty. "A Framework for Interconnecting Wireless Sensor and IP Networks." *Personal, Indoor and Mobile Radio Communications*, 2007. PIMRC 2007. IEEE 18th International Symposium.

[25] Ping Song, Chang Chen, Kejie Li and Li Sui. "The Design and Realization of Embedded Gateway Based on WSN." In IEEE *2008 International Conference on Computer Science and Software Engineering*, 2008.

[26] Qing Yang, Alvin Lim, Kenan Casey, Raghu Neelisetti. “An Empirical Study on Real-Time Target Tracking with Enhanced CPA Algorithm in Wireless Sensor Networks,” *Ad Hoc & Sensor Wireless Networks An International Journal*. vol. 7, no. 3–4, pp. 251–271, May 2009.

[27] Qing Yang, Alvin Lim, Kenan Casey, Raghu-Kisore Neelisetti. “An Enhanced CPA Algorithm for Real-Time Target Tracking in Wireless Sensor Networks,” *International Journal of Distributed Sensor Networks*. vol. 5, no. 5, pp. 619–643, September 2009.

[28] Qing Yang, Alvin Lim, Kenan Casey and Raghu-Kisore Neelisetti. “Real-Time Target Tracking with CPA Algorithm in Wireless Sensor Networks.” *IEEE SECON 2008*, 2008.

[29] Rashmi Bajaj, Samantha Lalinda Ranaweera and Dharma P. Agrawal. “GPS: Location-Tracking Technology,” *Computer*, vol. 35, no. 4, pp. 92–94, April 2002.

[30] S. N. Simic and S. Sastry. “Distributed environmental monitoring using random sensor networks.” In *Proceedings of the 2nd International Workshohp on Information Processing in Sensor Networks*, pages 582-592, Palo Alto, California, 2003.

[31] Salem Hadim and Nader Mohamed. “Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks.” In IEEE Computer Society Vol. 7, No. 3, March 2006.

[32] Yun Hu, Fengqi Yu and Ping Deng. “A Novel Data Query Method for Wireless Sensor Networks.” IEEE, 2007.