Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE **On the Optimization of Multiple Applications for Sensor Networks**

PRESENTED BY **Vikram Gupta**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

ADVISOR, MAJOR PROFESSOR — DATE 10/15/2014

ADVISOR, MAJOR PROFESSOR — DATE 10/15/2014

DEPARTMENT HEAD — DATE 10/17/2014

APPROVED BY THE COLLEGE COUNCIL

DEAN — DATE

On the Optimization of Multiple Applications for Sensor Networks

Submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

in

Electrical and Computer Engineering

Vikram Gupta

Bachelor of Technology, Electronics and Communications Engineering,
Visvesvaraya National Institute of Technology, Nagpur, India

Master of Science, Electrical and Computer Engineering,
Carnegie Mellon University

Carnegie Mellon University, Pittsburgh, USA

Faculdade de Engenharia da Universidade do Porto, Portugal

Fall 2014

For the love of science

Acknowledgements

> We are not going in circles, we are going upwards.
> The path is a spiral; we have already climbed many steps.
>
> Hermann Hesse, Siddhartha (1922)

Working on doctoral research and a dissertation requires long commitment with focus, motivation and perseverance. Most of the time, this endeavor is very exciting and intriguing, but at times, it can also be slow and full of struggles forcing one to ponder over metaphysical things like the meaning of life itself. It is with the unwavering support of many key people that one can succeed in such an undertaking. First and foremost, I would like to thank my advisors, Prof. Raj Rajkumar and Prof. Eduardo Tovar for their guidance at every step during my PhD. I consider myself fortunate to have had great learning experiences from both my advisors, and their drive and motivation towards research is something that I aspire for.

Prof. Eduardo has always been immensely helpful in all things related to research, academics or administrative issues despite his busy schedule, which also included answering several last minute review requests near deadlines. Eduardo's scientific acumen, enthusiastic approach to new research problems and ability to manage varied responsibilities is very encouraging for his students. Without the faith he showed in me, my life as a PhD student would not have been as fulfilling. In addition to being an ideal advisor, I would also like to thank Prof. Eduardo for his indispensable role in setting up and directing the CISTER lab with incredible facilities and a great research environment.

Working under the supervision of Prof. Raj has been a great learning experience for me in many ways. His practical approach to research, attention to detail, organization of thought-process and inquisitiveness are some of his many qualities that I hope to imbibe in myself. He always strives for

perfection and expects the same from his students, which is very inspiring. I will always be grateful to both my advisors for supporting me and giving me an opportunity to work on interesting problems with great people.

I would also like to thank Prof. Peter Steenkiste, Prof. Anthony Rowe and Prof. José Silva Matos for agreeing to be on my thesis committee. Prof. Peter was one of my earliest contacts in Pittsburgh when I started at CMU. I had the opportunity to spend some time with him while sightseeing in Porto, before I formally started the program. Those interactions with him helped me understand what a PhD and the life in the US means, and I got a first-hand overview of research at CMU. Prof. Peter has always been very helpful and encouraging during several interactions I had with him. Prof. Anthony has been a senior mentor to me even when he was a graduate student during the early part of my doctoral studies. My first research project was under his co-supervision and it was a great initiation to various aspects of research including experimentation, design of hardware and technical writing. Since then, Anthony has been a great mentor and a very helpful critic. I am particularly thankful to Prof. Matos for agreeing to be on the committee. He is always approachable, kind and helpful to students as an in-charge of the Electrical Engineering PhD program at FEUP.

I also extend my sincere thanks to several colleagues at Carnegie Mellon: Karthik Lakshmanan, Arvind Kandhalu, Junsung Kim, Gaurav Bhatia and Reza Azimi. Karthik was also a mentor as a senior student and he was always ready to help and guide in many research/technical challenges. The fact that a course project with Junsung and Aditi Pandya as my team-members and Karthik and Arvind as course-teaching assistants became a stepping-stone for my thesis topic is an acknowledgement of their contribution, and I thank them for this. Being with such colleagues is a learning experience by itself, and informal discussions with them during lunch or coffee breaks were stress relievers, motivators and also stress enhancers (near deadlines) at the same time.

My fellow researchers and students at the CISTER research lab are a great set of people and they have been supportive in many ways. I would like to thank Nuno Pereira for his help, guidance and his faith in me, especially for including me in the writing of a project proposal. In addition, I would also like to thank Prof. Luis Miguel Pinho for his valuable inputs in the sMapReduce project. There is a long list of people who I would like to thank as well. Vincent Nelis, Geoffrey Nelissen, Patrick Yomsi, Gurulingesh Raravi, Ali Awan, Dakshina Dasari, José Marinho, João Loureiro, Ricardo Garibay, Shashank Gaur and Borislav Nikolic provided several opportunities to brainstorm on

academic and non-academic issues. In addition to creating a great working environment in the lab, our social interactions helped create a nice work-life balance. Innumerable evening tea sessions, several dinners and board-games nights has made my time in Porto very memorable. I am also thankful to Shashank Gaur for his help in the Network-Harmonized Scheduling project and Maryam Vahabi for giving me an opportunity to work on a really interesting problem.

I would also like to thank my close friends from undergraduate years who have been very supportive and encouraging. Keshav Seshadri is a very close friend and we were flatmates for the duration of my stay in Pittsburgh. He has always been a very generous host to me whenever I visited Pittsburgh for short durations. It was Keshav who first led me to apply for the CMU-Portugal program. His organized style of working and attention to detail has always motivated me. Keshav, Kandarp Shah, Soumya Jain and Anand Srinivasan have been close friends for more than ten years now, and meeting with them instills energy whenever I feel low and need a little push. They have always been ready to discuss my research problems, and provided a great third-person's point of view.

My family's faith and confidence in me was a guiding light throughout the PhD. My family has always supported me and given me freedom to pursue my dreams, except only the frequent "When will you graduate?" questions. I am thankful to my mother for always inspiring me to excel in academics. My father always encouraged my wild and crazy ideas in my childhood and taught me to remain curious. He is not with us anymore, but I hope that I have made him proud. I am indebted to my younger brother, Manish, who actually played the role of an elder brother by letting me pursue my graduate studies while he took care of the family back home. My elder sisters, Parveen, Payal and Hittu have always encouraged me to achieve the best. I sincerely hope that I continue to meet their expectations. Finally, I do not have many words to express my love and gratitude for my wife, Kritika who supported me throughout and made my student life so comfortable, even if it meant cooking snacks for me in the middle of the night or staying up until very late to align her schedule with mine.

Last but not the least, I would like to acknowledge *Fundação para a Ciência e a Tecnologia* (FCT) for providing the funding through the CMU-Portugal program and the PATTERN project. The staff at CMU and ICTI has helped me out several times in working out various administrative details. In addition to the administrative support, Prof. José Moura, Prof. João Claro and Prof. Aurélio

Campilho have always been extremely helpful in answering my doubts and pointing me in the right direction for fulfilling both the CMU and FEUP requirements.

Abstract

Wireless Sensor Networks are an important example of networked embedded systems, and they have a key role to play in the development and materialization of the concept of an Internet of Things. Most current deployments of sensor networks are very application-specific and only target a particular goal. This negatively impacts the cost-effectiveness of the network, thereby reducing the incentive to deploy a sensing infrastructure in the first place. *Multi-purpose* sensor networks can overcome these limitations, where they can be used for more than one application simultaneously. To promote the role of wireless sensor networks as an infrastructure technology, support for multiple independent applications is essential, such that different users can concurrently submit their applications to accomplish diverse goals.

In this dissertation, we address various challenges that arise when the support of multiple applications is enabled on a sensor network, mainly with respect to application development, installation and execution. First, we propose a holistic programming framework called *Nano-CF* that is built on top of the Nano-RK operating system that allows independent users to deploy applications at a network level, and coordinates network activity. Then, we propose various approaches to reduce the resource consumption at different levels in a sensor network. Most sensor networking applications are designed for sampling one or more sensors, conduct signal processing on the sensed data, and communicate this data with other devices in the network. With more than one such application executing on the network, it is highly probable that some redundancies occur *across* applications. In this work, we identify and eliminate these redundancies through a compile-time approach that identifies the temporal overlap across the execution of the applications. Moreover, we augment the Nano-CF framework with a hierarchical task-assignment scheme that selectively eliminates the redundancies while simultaneously conforming to the resource constraints of the sensor node and the application requirements. Finally, we propose a network-level scheme called *Network-Harmonized Scheduling* that coordinates the packet transmissions in a simple and distributed way such that the radio can be used efficiently in a multi-hop network with multiple applications releasing packets periodically.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Among the various milestones achieved by humankind in its history since the invention of the wheel, there is no denying that the invention of semiconductor electronics and, subsequently, computers are among the ones that have had the most significant impact on the world as we see today. Beginning with the vision of an "Intergalactic Computer Network" by J.C.R. Licklider in 1963 [7], the capability of computers to interact with each other has further revolutionized our lives with the development and large-scale acceptance of the Internet. Moreover, *embedded computing systems*, which were traditionally designed to be application-specific and standalone computers now also have the capability to communicate among each other and to the Internet.

In principle, an embedded computer system[1] is a computer that is designed for a very specific application, interacts with its physical environment and is typically not suitable for general-purpose computing. Examples of such embedded systems include controllers in home appliances such as thermostats, washing machines or even dedicated computers in automobiles, airplanes and satellites. In recent times, however, the boundary between embedded systems and generic computers is getting blurred because of the increase in computational power and decrease in size and cost because of Moore's Law [8]. Modern mobile phones, also called *smartphones*, are a prime instance of "the merge" of embedded systems with communication capabilities and high computing power.

On the other hand, Moore's law has also led electronic devices to become smaller in size at such

[1] Embedded-computer systems are also more commonly referred to as *embedded systems*

a low cost and low power consumption that it is expected that almost every object around us can become a *smart* embedded system. Already, devices around us are becoming increasingly smarter and many of those devices have sensors to measure physical quantities that can allow them to be aware of the surrounding environment and take better decisions while interacting with that environment. Next-generation applications of such embedded systems is expected to involve high-degree of inter-networking of devices, possibly with an aim to accomplish the vision of *Internet of Things*.

In this view, *Wireless Sensor Networks (WSNs)* have been playing an important role both as a technology-incubator as well as a test-bed for the networked embedded systems of the future. 6LoWPAN [9] is a standard in this direction that can bring Internet Protocol (IP) connectivity to small embedded devices in a similar way as general-purpose computers of today are interconnected via the Internet. Despite wireless sensor networks being a very active research field and a number of very highly-rated conferences and journals catering to this area, most of the works are limited to laboratory test-beds with very little adaptation to real-world deployments. One of the main reasons behind this is that the deployments are typically very application- or goal-specific to the problem being addressed. In order to promote fast adoption of sensor networks for practical usage, they should be developed as a long-term infrastructure technology with support for multiple applications from independent users, rather than as application-specific deployments for short-term use.

Therefore, in this dissertation, we propose a multi-dimensional approach to deploy more than one application on a sensor network and simultaneously reduce the resource consumption at various levels. Such multi-application or multipurpose embedded systems are quite in contrast to the early vision of computers as evident from the following quote by J.C.R. Licklider [10]:

> *Men are noisy, narrow-band devices, but their nervous systems have very many parallel and simultaneously active channels. Relative to men, computing machines are very fast and very accurate, but they are constrained to perform only one or a few elementary operations at a time. Men are flexible, capable of "programming themselves contingently" on the basis of newly received information. Computing machines are single-minded, constrained by their "pre-programming".*

Through the technologies we built and the optimizations we propose in this dissertation, we attempt to revise this widely-held notion of application-specific and inflexible networked embedded systems into an infrastructure technology that can cater to different needs of many users simultaneously. In

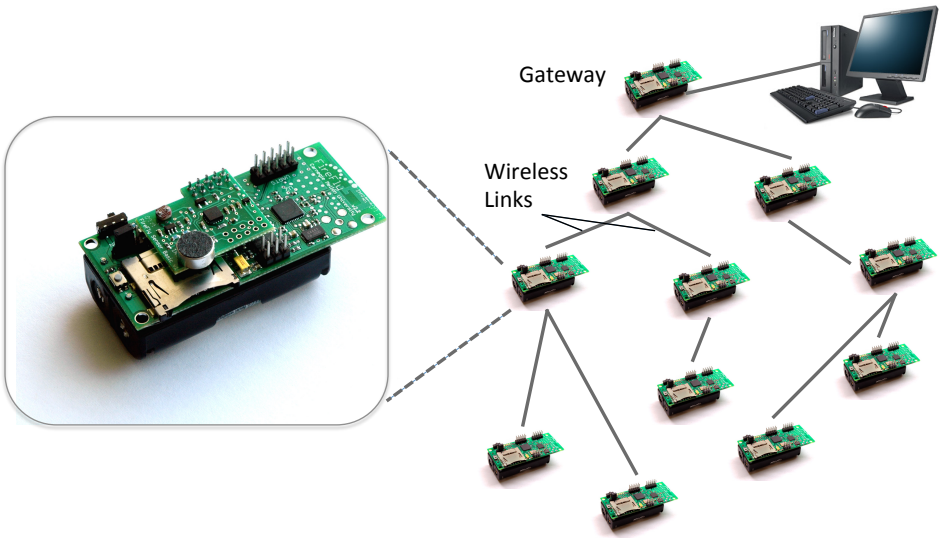


Figure 1.1: A typical sensor network with several Firefly [1] sensing platforms communicating over wireless links (shown with symbolic line-connectors). The network topology is a tree topology with a gateway node as the root. The gateway is connected to a PC, which acts as a router to the Internet.

this dissertation, we focus mainly on Wireless Sensor Networks (WSNs) as a representative technology.

1.1 Background on Wireless Sensor Networks

With the capability to communicate over wired or wireless links, embedded systems can create a complex ecosystem with an unfathomable potential, and Wireless Sensor Networks (WSNs) bring connectivity closer to the physical world by measuring and monitoring the physical quantities such as temperature, pressure, humidity and light intensity. This is made possible through inexpensive and small hardware modules with radio transceivers. Not only do wireless sensor networks have the potential to improve several day-to-day and fundamental systems such as transport, power distribution, agriculture etc., by enabling large-scale monitoring, they can also have an important role as test-beds or technology incubators for materialization of the concept of the Internet of Things.

Wireless Sensor Networks consist of a large number of tiny hardware devices called sensor nodes[2] that can form an ad-hoc mesh network so that the values sensed by individual devices can be collected at a sink (or gateway or root) as shown in Figure 1.1. Wireless Sensor Networks have been, and are currently being, used for a variety of applications and purposes that were inconceivable with traditional sensing methods of using large sensing stations requiring manual deployment and manual data-collection. Wireless Sensor Networks are designed to be operated autonomously once deployed, (potentially) without the need for any human intervention. There is a large variety of applications of sensor networks, and we briefly list some them below:

Environmental Sensing. Remote sensor networks have been deployed to monitor natural phenomena such as volcanoes [11], health of redwood trees [12] in California, US or underwater phenomena such as coral-reefs and fisheries [13, 14].

Structural Health Monitoring. Monitoring the health of civil infrastructures such as buildings [15], bridges [16], underground mines [17] or factory equipment [18] are some of the most popular applications of sensor networks.

Industrial Sensing. Industrial processes need to monitored regarding various parameters for smooth functioning of assembly lines and safety concerns [19, 20].

Understanding Social Behavioral Patterns. Sensor networks may enable the understanding of the social behavior with a human-centric approach where humans carry a portable sensor [21, 22]. Human health monitoring is also another example of human-centric sensing [23].

Disaster Management. Sensor networks are helpful in disaster situations such as forest-fire monitoring [24] or search and rescue operations [25].

Building Monitoring. There are several other applications that help in accomplishing smart-buildings and reducing the energy consumption. For example, building automation [26], energy-audit [27] and data-center monitoring [28].

In this dissertation, we focus on sensor networking applications mainly related to building monitoring. In such applications, various sensors are permanently deployed inside buildings to observe

[2] A more detailed description about various platforms used as sensor nodes is provided in Chapter 2

energy consumption, temperature, occupancy, humidity and noise etc., and different users would benefit in using that infrastructure to run their independent applications simultaneously.

1.2 Motivation and Challenges

Modern sensor networking platforms are complex devices with multiple types of sensors, allowing independent users to employ one or more sensors on each device for creating network-level applications. Let us consider the example of a university campus sensor network deployment, such as the Sensor Andrew [29] project at the Carnegie Mellon University. Sensor Andrew is deployed across several buildings to monitor physical properties of the environment such as temperature, humidity, smoke or light. A diagram of a sample building with a variety of networked sensors is shown in Figure 1.2. To facilitate the appropriate use of such a network, it is beneficial to provide support for multiple applications from users with different goals and backgrounds such that the network infrastructure can be treated as a *multi-purpose sensor network* [30]. For example, a building manager may be interested in collecting the temperature values from the sensors for a fine-grained temperature control, while a civil engineer may want to find the correlation between temperature and humidity for optimizing a building's HVAC system. Furthermore, a security officer may want to monitor the fire alarms periodically. With multi-purpose sensor networks, several varied applications may be concurrently supported on the same infrastructure. In this dissertation, we address various challenges that emerge from allowing multiple applications to execute on a shared sensor network infrastructure.

There are several advantages in enabling such multi-purpose sensor networks, including improving cost effectiveness, enlarging the user-base of the sensor networks, and enabling the integration of sensor networks with other large-scale technologies. With multiple applications, the sensor network infrastructure can be utilized better, in a cost-effective and in a collaborative manner by independent users. Moreover, multi-purpose sensor networks can provide seamless integration with larger-scale technologies such as the Smart-Grid or the vision of Internet of Things. As an example, power companies may also want to deploy suitable applications on sensing devices installed in buildings so that they can access *(limited)* data about usage patterns of various appliances. This data can help the power companies to schedule load-balancing strategies with better efficacy. In addition to the advantages from the application perspective, multi-purpose sensor networks also encourage users from diverse

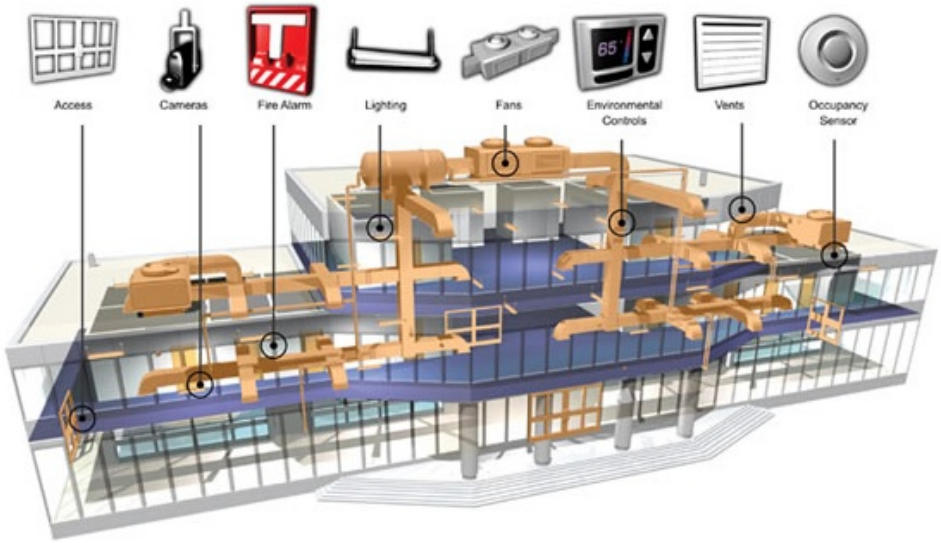


Figure 1.2: An example of an indoor sensor network with various types of sensors deployed at different locations inside a building. Sensor Andrew is an example of such a network, but spread across multiple buildings.

backgrounds to develop applications in a convenient manner, thereby promoting multi-disciplinary research.

In spite of the advantages discussed earlier, several challenges arise with multiple applications executing on sensor networks, mainly due to the resource-constrained nature of sensor nodes. As the primary design principle behind sensor nodes is to keep the hardware cost low and to attain long-life with relatively cheap batteries, reducing the average energy consumption has been one of the foremost challenges in this area of research. With multiple applications, several new challenges arise that need to be addressed for the proper management of the network resources, and are discussed below.

1.2.1 Programming Support

Programming sensor nodes individually is a time-consuming and complex procedure, so several macro-programming approaches and middleware have been proposed in the past to install applications at the network-level, such as Regiment [31], Mate [32] and TinyDB [33]. Except for a few

solutions proposed recently [3], not many approaches support deployment of multiple concurrent applications at a network-level. Considering the example of Sensor Andrew, users from diverse technical backgrounds may benefit from a middleware layer that helps them program the network at an abstract higher level. A conventional macro-programming framework may not allow more than one user to independently program the sensor network for handling these kinds of applications simultaneously. To support multiple applications, several additional aspects have to be managed regarding programming abstractions, application installation, execution and data delivery.

1.2.2 Inter-application Redundancy

Most sensor networking applications are designed with the goal to sample one or more sensors, conduct some signal processing on the sampled data, and then share it with other nodes or a base-station to accomplish a distributed logic. In the case of more than one such application, there is a likelihood of redundant sample collections by different applications causing unnecessary energy consumption. It is observed that reading a sensor value typically involves accessing an Analog-to-Digital Converter (ADC) module on a microprocessor, for converting the analog sensor value into a digital format, and storing into a register or local memory. This process of sampling a sensor can consume about 2–3 orders of magnitude more processor cycles than a simple memory-based instruction, thus increasing the processor usage on a sensor node.

1.2.3 Conforming to Resource Limitations

With more than one application deployed on sensor nodes, various resources tend to be used more frequently. Finding a balance between the application requirements and the available resources is a challenge. There is a need to optimally assign resources among applications, such that application requirements are met and the overall resource consumption is reduced as much as possible.

1.2.4 Network Coordination

In addition to resource optimization on individual nodes, several issues still exist in the networking layer when multiple applications execute on sensor nodes. It is often the case that applications release

packets independently in the network, which can lead to excessive energy consumption because of several factors such as an increase in the number of packets, more frequent radio-switching and increased contention at the Medium Access Control (MAC) layer. Energy consumed in transmitting a packet from a source node to a destination node depends on many aspects, and with common MAC approaches, a packet may undergo contention at several points in a multi-hop network, significantly adding to the overall radio-usage.

1.3 Thesis Statement

Although sensor networks are a popular technology as a research platform, their widespread adoption in real life is still limited because of their application-specific nature. Supporting multiple applications can help in making sensor networks a multi-purpose infrastructure technology. Deployment and execution of multiple applications come with challenges of complex programming, non-optimal resource consumption and uncoordinated network behavior. These challenges can be addressed by designing a holistic framework that helps in programming, deployment and management of applications, and includes optimizations for redundancy elimination across applications and a network-harmonization protocol.

In sum, the thesis of this dissertation is:

> *A holistic framework with suitable optimizations can be designed for the deployment and execution of multiple applications on a sensor network infrastructure without compromising the resource usage and the overall energy consumption of the network. The resource consumption at each device can be optimized by eliminating any redundant sensor-sampling requests across applications, and energy-efficient communication can be achieved by harmonizing packet transmissions.*

In this work, we propose a multi-dimensional approach that, in addition to facilitating the development and deployment of applications on a sensor network, also provides several solutions to reduce the resource consumption at both the node- and the network-level in *multi-purpose sensor networks*.

1.4 Proposed approach

In this section, we provide an overview of our proposed approach that addresses the challenges discussed previously.

1.4.1 Programming Framework

We propose the design of an over-the-air programming framework for sensor networks called Nano Coordination Framework (Nano-CF) [34]. Nano-CF allows multiple users to deploy their network-level applications on a sensing infrastructure independently without any interference from other applications. Our framework is based on Nano-RK [35], a Real-Time Operating System (RTOS) developed at Carnegie Mellon University. Nano-CF includes a programming abstraction, network-level application management and runtime layer for each node. The resource-kernel properties of Nano-RK are exploited by Nano-CF such that the resource requirements of each task (for example, processor utilization) can be declared at design-time and the kernel guarantees that an application does not consume more resources than its allotted share.

An overview of the Nano-CF programming framework proposed in this work is presented in Figure 1.3. In addition to the simple programming abstraction included as a part of Nano-CF, we also propose a design pattern, *sMapReduce* [36] inspired from Google's MapReduce concept. With sMapReduce, the programmer writes the software in two parts: *sMap* defines the functionality to be carried out by each device, and *Reduce* defines the data aggregation logic over the network.

1.4.2 Redundancy Elimination

Applications deployed on a sensor network infrastructure by independent users have the possibility of redundant accesses to the same sensors to collect samples. It is observed from experiments that sampling a sensor takes several orders of magnitude more processor cycles than using a value stored in local memory. By sharing sensing requests among applications, a significant percentage of resource usage and energy can be saved on a sensor node. We propose a novel solution [37] to the problem of finding redundant sensing requests issued by network-wide applications created by independent users. We model each application as a linear sequence of executable instructions, and find a

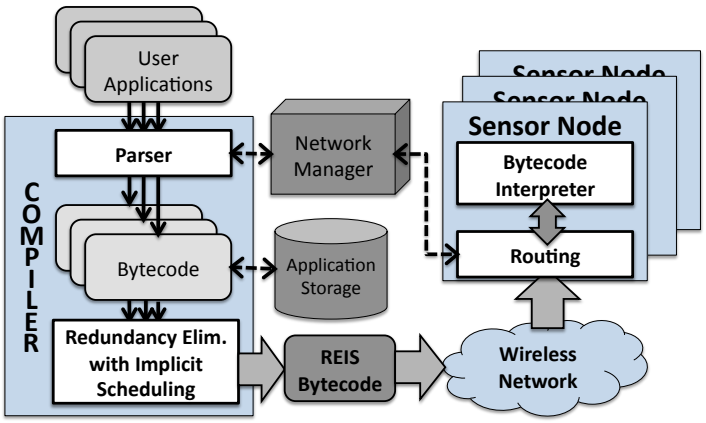


Figure 1.3: An overview of the Nano-CF programming framework with inter-application redundancy elimination (REIS) support.

merged sequence from them through the use of well-known string-matching algorithms. In particular, we use the Longest Common Subsequence (LCS) [38] and the Shortest Common Super-sequence (SCS) [39] techniques. Our proposed solution, called *Redundancy Elimination with Implicit Scheduling (REIS)*, creates a monolithic *task-block* resulting from an optimized merging of user applications with embedded scheduling information. REIS is a compile-time tool and is a part of the compiler in the overall framework shown previously in Figure 1.3. This scheme is particularly advantageous in cases where the relative order of sensing requests is important, and simply caching the values may not help. We show that our approach can help in achieving significant average energy savings in processor usage as compared to the execution of several applications without eliminating the redundancies. Moreover, the radios on modern System-on-Chip solutions for sensor networks such as the Atmel ATmega128RFA1 [40] can transmit at 8 times the data rate for the same amount of power as earlier designs. This means that the power-per-packet metric can be reduced by a corresponding factor. Hence, optimizations at the processor level are bound to play a significant role in reducing the total energy consumption, in contrast to the majority of the research efforts focusing mainly on energy savings at the radio-level.

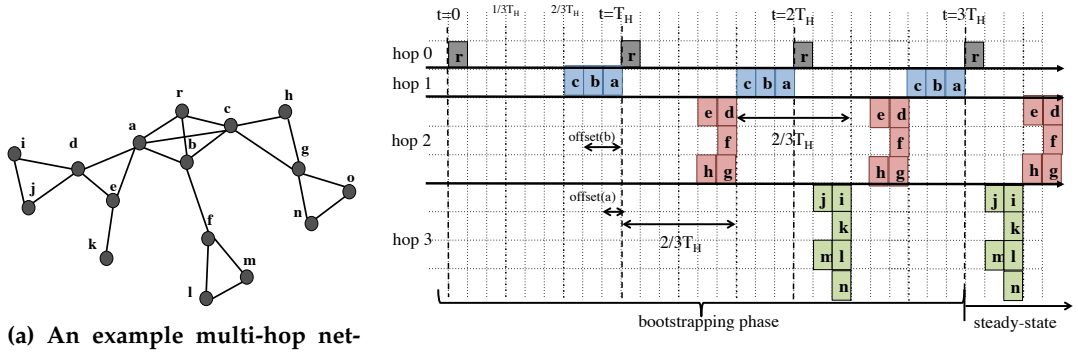
1.4.3 Hierarchical Assignment

To maximize the energy savings achieved through redundancy elimination while meeting the resource constraints on sensor nodes, we propose a hierarchical approach where sensing redundancies across a subset of applications are eliminated. The goal is to strategically assign user-designed tasks into multiple task-blocks such that several application requirements can be met. For example, merging applications with differing periods can cause the task-block to have a large memory footprint, and it may be beneficial to create more than one task-block such that the tasks with similar periods are together. Similarly, creating task-blocks from input tasks which share the same sensors may be a better approach than from tasks that sample different sensors.

Our proposed hierarchical scheme addresses varied challenges, ranging from the modeling of applications to the formulation of the optimization problem. Firstly, a maximization function for energy savings is devised, which depends on the degree of redundancy across a set of applications. The degree of redundancy, however, is found using string-matching algorithms that makes it impossible to model the problem in a standard formulation solvable by commercial solvers. We then present a solution that decouples the assignment optimization problem from string matching. Finally, the problem is reduced to that of Quadratic Integer Programming, which can be solved using well-known approaches.

1.4.4 Network-Harmonized Scheduling

More than one application on sensor nodes can result in irregular packet-transmission behavior even if each application is periodic. This can result in additional latency and power-consumption because the packets may experience contention more frequently at different stages in a multi-hop network. We propose a scheduling approach, partly inspired by Rate-Harmonized Scheduling [41], that aligns the packet transmissions from a sensor node along a periodic boundary. We call our approach *Network-Harmonized Scheduling (NHS)* [42]. NHS includes a light-weight protocol that groups periodic batched transmissions from different devices, such that the nodes can turn on their radios when other devices transmit. The operation of the NHS protocol in an example multi-hop scenario is shown in Figure 1.4, with 12 nodes and the corresponding timeline of packet transmissions. The nodes at successive hop-levels transmit periodically, but with an offset so as to avoid any packet-loss due to



(a) An example multi-hop network, with 12 nodes, and a root node (r).

(b) Timeline of transmissions from nodes at different hops, showing the working of the NHS protocol.

Figure 1.4: An example of a multi-hop topology, and the corresponding timeline for illustrating the NHS protocol within a network.

the hidden terminal problem. The main goal of this protocol is to enable scheduled transmissions in a distributed manner, without requiring global knowledge of the network topology and without explicit time-synchronization. The working principle behind the protocol is to ensure more than a 2-hop distance in simultaneous transmissions in the network. In principle, NHS can achieve a TDMA-like duty-cycling with very little state maintenance and without the help of a dedicated central controller. NHS shows that it is possible, and beneficial at the same time, to coordinate network access across multiple hops in a simple and efficient manner. This approach provides determinism in the network operation, and certain guarantees can be made about the performance of the protocol.

After outlining our overall approach in this dissertation, we now provide the scope of our research.

1.5 Scope of the Dissertation

Wireless Sensor Networks is a large research domain where researchers have addressed various challenges from varied perspectives including, but not limited to, hardware design, operating systems, networking, signal processing, security and data-mining. In this dissertation, we specifically address issues that arise with multiple applications executing on a sensor network.

Depending on the requirements of the applications, wireless sensor networks can have different physical and logical deployments. Infrastructure deployments are long-term deployments composed

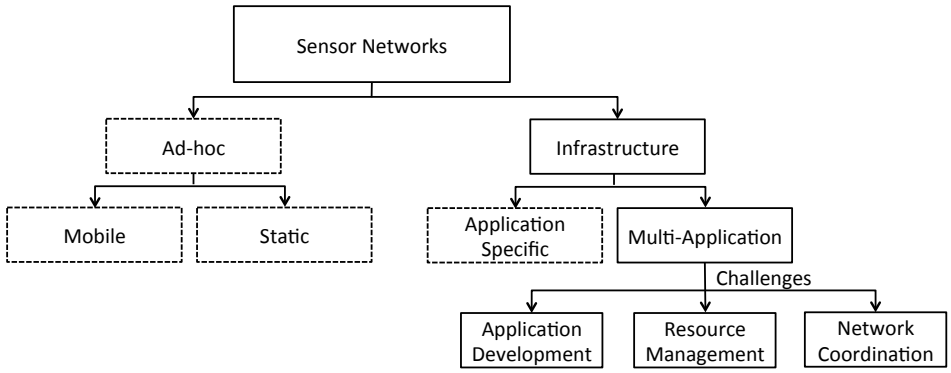


Figure 1.5: The figure depicts the scope of this dissertation. The solid boxes represent the target research domain.

of stationary nodes deployed typically inside buildings for observing their indoor environment or monitoring their structural health. Ad-hoc networks, on the hand, can consist of mobile or static nodes with intermittent connectivity deployed in remote areas or urban short-term deployments for experimentation purposes. In the scope of this dissertation, illustrated in Figure 1.5, we address the challenges of application deployment, inter-application redundancy elimination and network coordination that arise with multiple applications running on stationary infrastructure sensor networks.

1.6 Organization of the thesis

The rest of this dissertation is organized as follows:

- In the next chapter, we provide a background into various technologies employed in this dissertation, including hardware platforms, operating systems and communication protocols. This chapter serves two purposes: providing an understanding on various issues in a sensor network from different perspectives and describing briefly the platforms and protocols based on which our approaches are built.

- We provide a detailed description of the related work and literature survey in Chapter 3. We describe the state of the art and other research that address similar challenges, and also contrast them against our approach.

- In Chapter 4, we describe our proposed Nano-CF programming framework to support multiple applications on sensor networks. Nano-CF allows a programmer to write a network-level application and the framework transparently handles the application delivery and the runtime executes it on each node on top of the Nano-RK operating system.

- In Chapter 5, we present a compiler optimization approach based on the observation that independent applications developed in Nano-CF may issue redundant sampling requests leading to the over-consumption of resources. We propose the design of a compile-time approach called REIS that finds maximum overlap *across* applications such that redundancy is eliminated.

- We then extend the REIS approach further with a hierarchical assignment described in Chapter 6 scheme that instead of finding redundancies across all the applications, finds it across a subset of applications such that the redundancy elimination is maximized while satisfying the resource constraints of sensor nodes.

- While expanding the focus from node-level optimizations to network coordination in case of multiple applications, we propose Network-Harmonized Scheduling in Chapter 7 that aligns the packet releases from multiple applications along periodic boundaries, and then harmonizes them across the network.

- Finally, in the last chapter, we conclude this dissertation with a summary of contributions and a discussion of future work. As a part of future research directions, we also provide a vision of a new operating system paradigm for sensor networks called *Co-Operating System* that is distributed by design and allows application development at a larger scale than node-level.

Chapter 2

Background

As we briefly described in the previous chapter, wireless sensor networks consist of a large number of sensing devices connected to each other via wireless links. Wireless connectivity eases the deployment of a sensor network, as laying wires for a large number of devices can be unpractical and expensive both in terms of cost and man-hours. With wireless communication support, the devices can be deployed in remote terrains with relative ease and old buildings can also be retro-fitted without the need for re-wiring. Furthermore, the use of batteries to power wireless devices is a logical choice to completely remove the need for wiring. On one hand, batteries and wireless communication enable sensor networks to be used effectively with an ease of deployment, but on the other hand, battery-powered and wireless operation are two of the most important factors that bring forth a broad range of challenges in all the aspects of sensor networking.

In this chapter, we briefly outline some of the many fundamental technologies that are employed in sensor networks to address the challenges that mainly are manifestations of battery-powered and wireless operation. Throughout the research efforts presented in this dissertation, we have made use of a majority of these technologies that include hardware platforms, operating systems, programming frameworks and networking protocols.

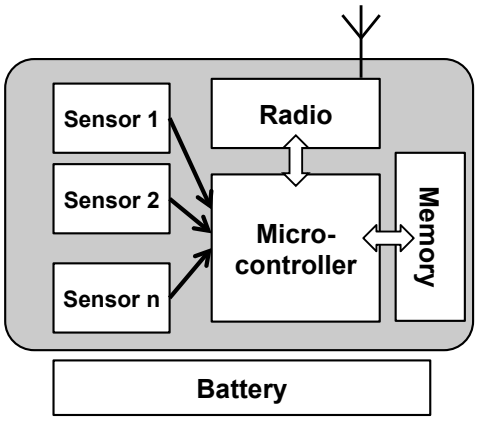


Figure 2.1: A schematic diagram of a sensor node

2.1 Platforms

Several hardware platforms have been developed in the past as prototype devices for wireless sensor networks, and many such devices are commercially available. Some of the popular choices for general-purpose sensor networking platforms include Mica2 [43], MicaZ [44], Sun SPOT [45], TelosB [46] and Firefly [1]. There are several other experimental platforms that serve application-specific goals, such as energy-harvesting [47, 48], structural monitoring [16], industrial sensing [19], data-center monitoring [28, 49] or smart homes [50]. In this dissertation, we limit our description to general-purpose sensor platforms, as our solutions are built for such devices. Various terms are used interchangeably to refer to a sensing platform as explained next:

> Individual hardware devices in sensor networks are typically referred to as *motes*, and in the larger context of a network of interacting motes, each device is also referred to as a sensor node, or more commonly as a *node*. Even though a sensor node consists of one or more sensors, a microprocessor and a radio transceiver among several other peripherals, the term *sensor* is also used interchangeably for a *sensor node*, as one of the key jobs of a node is to sense the environment and share it with other devices.

Sensor nodes are highly energy constrained devices, powered either with the help of batteries or energy harvesting systems such as solar panels. As a result, sensor nodes are designed with highly resource-constrained hardware components due the limitations of the power-source. A typical sen-

Table 2.1: Comparative specifications of micro-controllers commonly used in sensor nodes

Parameter	TI MSP430[a]	Atmel ATmega1281	Atmel ATmega128RFA1
Architecture	16 bit	8 bit	8 bit
Max. Frequency	16 MHz	8MHz	16MHz
RAM	10KB	8KB	16 KB
Flash	48KB	128KB	128KB
Sleep Current	5.1μA	5μA	5μA
Active Current[b]	1.8mA	10mA	4.1mA

[a] Specifications for MSP430F2611 used in TelosB motes
[b] at 8MHz and 3V

sor node consists of a low-powered micro-controller that is connected to a relatively simple and low-powered radio-transceiver. Several different sensors for monitoring temperature, pressure, humidity, acceleration, motion, etc., are also provided on board, and almost all platforms have expansion ports to connect to other specialized sensors. A schematic diagram of a typical mote with its components is shown in Figure 2.1. Firefly and MicaZ motes use the Atmel ATmega1281 micro-controller [51], and TelosB motes, on the other hand, use the Texas Instruments (TI) MSP430 [52] micro-controller. Both these micro-controllers are low-powered and have an 8 bit architecture with on-chip memory and flash. The Atmel ATmega1281 micro-controller has only 8 Kilo Bytes (KB) of RAM and can operate at a maximum frequency of only 8 MHz. The capacity of the on-board flash to store the firmware is about 128 KB, which means that the entire software stack should fit within this limited storage. These specifications are several orders of magnitude lower than that for a modern personal computer or even a modern handheld smartphone. The Moore's law is expected to help in reducing the size, power requirements and cost rather than improving the computational capabilities. Newer designs of motes, like the Firefly version 3, use System-on-Chip (SoC) solutions such as the Atmel ATmega128RFA1 [40] that has the processor and the radio transceiver on the same chip, thereby reducing the size of the motes and allowing more power-efficient designs. Figure 2.2 shows a snapshot of the latest Firefly mote with an expansion board consisting of a variety of sensors. Table 2.1 lists some relevant specifications of TI MSP430, ATmega1281 and Atmel ATmega128RFA1, to emphasize the limited memory and computational power available with sensor nodes.

Wireless communication among the nodes in a network is achieved using a low-power Radio-Frequency (RF) transceiver such as the Chipcon CC2420 [53]. Even if designed for low-power opera-

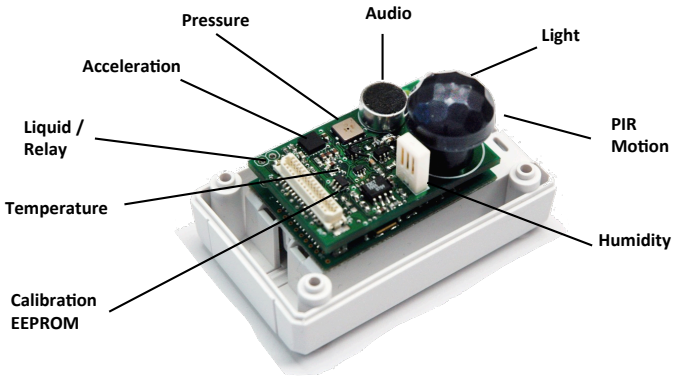


Figure 2.2: A Firefly version 3 node with a sensor expansion board.

tion, the radio is still the single largest energy consuming peripheral on a sensor node. In most cases, the power consumption by the radio can be up to 3-4 times compared to that of the processor. For CC2420, active current drawn during the transmission of a packet is 17.4 milliamperes (mA) and is counter-intuitively lower than that for reception, which is 19.7 mA. The higher energy requirements of the radio make it important to design efficient communication-layer and networking mechanisms for sensor networks.

If a sensor node is powered with a pair of off-the-shelf AA batteries of about 2000-3000 mAh capacity each, a continuous (always on) operation will yield only a battery life of about 2-3 days. This would mean that the batteries would need to be changed or recharged very frequently, making remote and/or large sensor network deployments impractical. In order to increase their lifetime, sensor nodes are typically operated with a very low-duty cycle of the order of 1% or 0.1%, such that sensor nodes remain in the sleep mode for a majority of the time. In this way, the sensor nodes can operate for a longer duration up to several months with the same batteries. Table 2.2 lists the current drawn and the corresponding battery lifetime for a Firefly sensor node under various operation states, when powered by two 2750 mAh AA batteries. Reducing the communication overhead and maintaining a very low duty-cycle are primary concerns while designing the software layers of sensor networks aiming for a long operational-lifetime with small and inexpensive batteries. Hence, miniziming the energy consumption and improving the energy-efficiency is a fundamental principle based on which viable sensing systems are built.

Table 2.2: Typical current drawn and corresponding battery lifetime for a Firefly [1] sensor platform under various operation states.

Firefly Operation State	Current Consumption	Battery Life
Processor Active	7 mA	17 days
Active with Transmission	24.4 mA	4.9 days
Active with Reception	28.8 mA	4.6 days
Idle	2 mA	60 days
Sleep	70 μA	**4.68 yrs**

2.2 Sensor Network Software Architecture

Sensor networking platforms such as the MicaZ and the TelosB, mentioned in the previous section, are highly resource-constrained. However, they are complex devices with several peripherals that can easily support a small operating system, custom communication stacks and one or more software applications. Programming the sensor nodes by directly writing the application logic on the system flash (colloquially referred to as *bare-metal* programming) without an operating system can be a daunting task even for skilled developers. To circumvent this challenge of developing software for sensor networks, suitable programming support has been developed in the form of various operating systems and network-level programming abstractions, as discussed next.

2.2.1 Operating Systems

One of the most important software component in sensor networking is an Operating System (OS), which provides several convenient abstractions to ease the application development. An OS also handles key responsibilities such as scheduling the tasks, process management, interfacing with hardware through device drivers and power management. In comparison to general-purpose computers, where operating systems handle several additional responsibilities such as memory management, virtual memory, graphic user-interface management etc., sensor networking operating systems [54, 55] are limited in features due to the highly resource constrained hardware of sensor nodes. The application-specific nature of sensor networks results in a wide variety of the underlying hardware, and therefore, operating systems play an important role in hiding this heterogeneity from the applications. Consequently, operating systems are imperative for providing hardware abstraction,

management of the limited resources and enabling communication with the other devices in the network.

Operating systems for sensor networks are designed to satisfy several different requirements, and some of them are listed below:

- **Providing Hardware Abstraction**: One of the foremost goals of an operating system is to provide abstractions to the underlying hardware, and it becomes even more important for sensor networks due to their heterogenous nature. Suitable software modules and device-drivers are included in the OS for sensor nodes to facilitate the interaction with peripherals such as sensors, actuators and the radio transceiver.

- **Power Management**: As sensor nodes are powered using batteries, they have a limited energy source and managing the energy requirements becomes another important responsibility of the operating system. In addition to providing explicit power-management control to applications to turn peripherals off when not in use, the operating system can also implicitly turn-off the unused devices even when applications are not designed to do so.

- **Resource Management**: Sensor nodes are resource constrained in terms of processing power, memory size and networking bandwidth, so the operating system ensures that resources are fairly used among various tasks. If some cases, the operating system can also enforce fair resource-usage by preempting a misbehaving task.

- **Handling Communication**: The operating system provides a configurable and modular communication stack to enable the applications to interact with the other devices. To achieve this goal, operating systems include Application Programming Interfaces (APIs) to allow programmers to use various communication protocols with ease.

- **Task Scheduling**: The operating system is also responsible for scheduling the tasks on a sensor node while considering the application requirements such as minimizing latency, ensuring fairness and maximizing the throughput. Applications with real-time constraints may require a real-time scheduler and the operating system can provide support for such a scheduling mechanism.

Various specialized operating systems have been developed in the past such as TinyOS [56], Con-

tiki [57], Mantis [58], LiteOS [59] or Nano-RK [35]. There OSes address the requirements listed above in different ways.

TinyOS [56] is one of the first and one of the most popular operating systems that is designed for sensor nodes. TinyOS is small in size (about 400 bytes kernel) and provides a component-based interface for specifying the interaction of various devices. A TinyOS configuration consists of the application, the corresponding operating system services and the scheduler in form of components. All the components are glued together to form a static/monolithic image that runs on a sensor node. A TinyOS component is formed with a composition of three abstractions: commands, events and tasks. Commands are used to trigger other services and events, in turn, signal their completion. In this way, TinyOS has an event-driven design with asynchronous events to handle computational entities that are called *tasks*. Early versions of TinyOS had a simple run-to-completion based First-In First-Out non-preemptive scheduler. A newer version of TinyOS with TOSthreads [60] supports a threading model with more flexibility. TinyOS uses the NesC [61] programming language for the specification of components and their concurrency model.

On the other hand, the Nano-RK [35] operating system has a time-triggered and event-triggered execution model. The main highlight of Nano-RK is its prioritized resource-kernel [62] approach where the kernel can enforce resource-usage limitations upon different tasks in the system. For example, if a task uses more processor cycles than its pre-allotted share, then the kernel can preempt the task and allow its execution only in its next cycle.

> The resource-kernel model of Nano-RK is capable of ensuring fair-sharing of resources among competing tasks; that is why we use Nano-RK as the base operating system for building the Nano-CF framework for multiple network-level applications on a sensor networking infrastructure as described in Chapter 4.

Contiki [57] is another popular operating system with a large user-base and support for the 6LowPAN [9] networking standard. Contiki has an event-driven execution model like TinyOS, but has a modular design with support for dynamic loading/unloading of modules. Contiki supports both synchronous and asynchronous events, that helps in achieving a better timeliness behavior where synchronous events get scheduled immediately and asynchronous events follow a queue. Contiki also supports a rich set of communication protocols including various medium access and higher-

layer network protocols. We have used Contiki in this dissertation for the implementation of the Network-Harmonized Scheduling protocol described in Chapter 7, mainly because it supports varied hardware, supports full Internet Protocol (IP) connectivity and makes network simulation convenient with the COOJA [63] simulator.

2.2.2 Programming Support

Programming a new hardware device is typically a significant deterrent particularly for researchers not familiar with embedded systems programming, and the difficulty increases several-fold if the devices need to communicate, for instance, to implement a distributed application. In the context of embedded systems and wireless sensor networks, usability and programmability are closely related as the user interaction with the devices happens mainly via programming interfaces rather than a graphic user interface as in the case of general-purpose computers.

The most basic way of programming sensor nodes is to connect them individually to a computer via a serial connection and uploading an application-specific firmware. Programming a large network in this manner can be very time-consuming, as it may involve collecting all the sensor motes from the field of deployment, programming them and then deploying them back. This method of programming is referred to as Manual Programming in the classification we provide later. On the other hand, to facilitate a faster and large-scale application deployment, it is useful to program the network as a whole at a higher level. This mechanism of programming the network at a larger- or macro-scale is referred to as *macro-programming*. The spectrum of macro-programming approaches range from delivering application-specific virtual machines to individual nodes like in Matè [32] to an abstract high-level programming like Logical neighborhoods [64] or Regiment [31]. Typically, the ease of programming increases with an increase in abstraction but it comes at the price of flexibility. We, therefore, divide the different programming approaches into following classes:

- **Manual Programming**: Programming sensor nodes over bare metal or an operating system by manually connecting a sensor node to a computer and flashing a firmware.

- **Over-the-Air Virtual Machine**: Creating a node-level virtual machine and delivering the binary or byte-code to the nodes over-the-air.

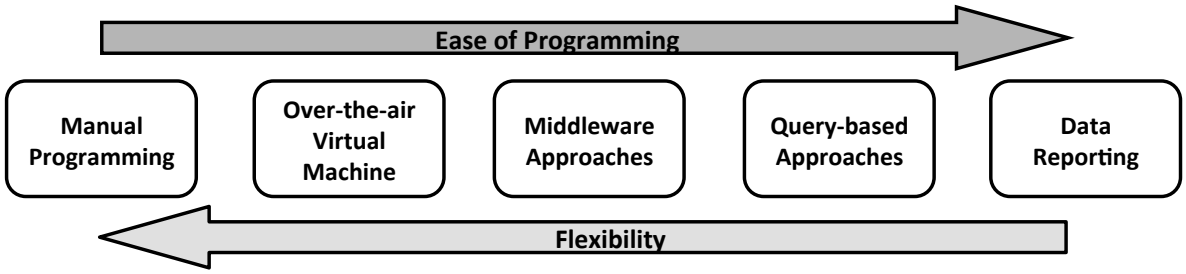


Figure 2.3: Ease of programming vs. flexibility in various classes of programming approaches.

- **Middleware**: Allowing the development of network-level applications as one application while hiding the network complexities.

- **Query-based approaches**: Writing SQL-like queries to request simple/aggregated data from the network.

- **Data Reporting**: Storing all the sensor readings periodically in an external database and accessing them when required. It is typically used in high-level Service-oriented Architectures (SoA).

Figure 2.3 shows different classes of programming methods for sensor networks. The ease of programming increases from left to right, but the flexibility follows the opposite direction. Most macro-programming schemes are limited to application-specific network operation, and our proposed framework called Nano-CF (described in Chapter 4) supports multiple independent applications with several resource optimizations at the node- and network-level. A more detailed discussion of the various state-of-the-art macro-programming approaches is provided in Chapter 3.

Reprogramming the devices or adding new applications *after* the devices have been deployed is still an active research challenge, and require a paradigm shift in the way devices are programmed and applications are installed. This is particularly challenging because the number of sensor devices and their ability to interact has been steadily increasing. Hence, providing the support for dynamic application deployment and its standardization is of paramount importance for increasing the outreach of *smart* sensing devices to diverse users, much like the concept of app-store has revolutionized the way we use mobile phone today.

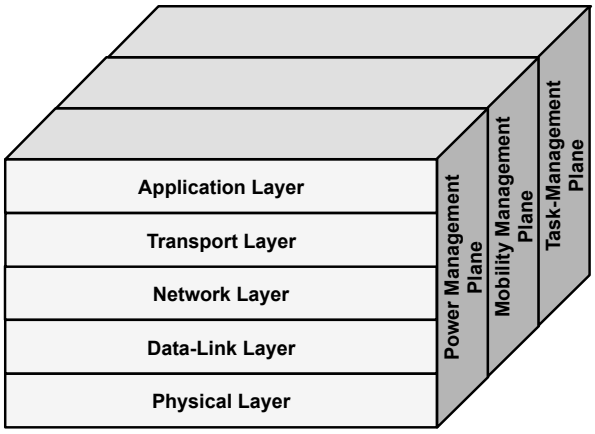


Figure 2.4: A generic protocol stack for sensor networks.

2.3 Communication and Networking

As mentioned earlier, radio usage is the most energy-consuming process on sensor networks and designing efficient communication mechanisms is of paramount importance to reduce the overall power consumption. Existing communication protocols such as Bluetooth [65] and Wifi [66] are not directly applicable and require a radical design-rethinking at all the layers of the communication stack. A widely-accepted and generic layered structure for communication in sensor-network proposed by Akyildiz et al. [67] is shown in Figure 2.4, that covers various layers starting with an application layer at the top and a physical layer at the bottom. This protocol stack is similar to the Open-System Interconnect (OSI) model [68], but it also includes some specific management planes for power management, mobility management and task-management for sensor networks.

In sensor networks, the application layer can consist of one or more network-level sensing and maintenance tasks such as data-dissemination [69, 70], network security [71], localization [72] and also time-synchronization [73, 74]. The transport layer is responsible for maintaining the flow of data from a source to a sink node, and the network layer is tasked with routing of data-packets over a multihop network. The data-link layer is responsible for negotiating the medium access for each node with as little energy-consumption as possible. The bottom-most physical layer in sensor networks is typically accomplished using a low-power radio transceiver that can operate with a few milliamperes of current consumption. The use of a simple modulation scheme with spread-spectrum techniques and lower data rates are some of the design options that allow a low-powered physical-

layer operation in sensor networks.

Sensor network deployments can be spread over large areas so it may not be practical to have a star-topology where a resource-rich access point interacts with mobile stations. A mesh-network (as shown previously in Figure 1.1 and later in this chapter in Figure 2.7) is more feasible where all the devices participate both as data-sources and routers. The nodes in such a network can establish intermittent connections with other devices in case of frequent/nomadic mobility. On the other hand, stable and long-term links can also be established for relatively stationary devices as in the case of building/infrastructure monitoring applications. Such a mesh-network is also called an ad-hoc network as various devices can communicate among each other for short or long durations, depending on the mobility patterns. In order to ensure the interoperability among the diverse sensor networking devices, several communication standards have been proposed, and have already gained significant popularity. The Institute of Electrical and Electronics Engineers (IEEE) has defined the IEEE 802.15.4 [75] standard for low-power Personal-Area Networks (PAN), that covers the physical and medium-access layers. ZigBee [76] is a corresponding industry consortium that implements and promotes the IEEE 802.15.4 standard. With wide availability of ZigBee compliant low-power devices, IEEE 802.15.4 has become a *de-facto* standard specifically for the physical layer in sensor network deployments and test-beds.

We now describe some data-link and network layer protocols that relate to the approaches proposed in this dissertation.

2.3.1 Medium-Access Layer Technologies

Radio transceivers on sensor nodes are designed to be low-powered, but even listening for packets costs energy, and this makes it impractical to have an *always-on* operation to wait for incoming packets. So the medium access layer has to be designed in such a way that efficient communication can be accomplished even if a listener node is off for the majority of the time and is not receiving packets. Secondly, in a shared medium implemented with wireless channels, a listener cannot receive packets from more than one transmitter at the same time because the packets can *collide* in the medium, and the receiver may not be able to correctly decipher the data from either packet. Although, some packets can be received correctly due to the *capture effect* [77], but in general, collisions lead to packet loss in wireless networks. Thirdly, radio transceivers are simplex devices *i.e.*, it is typically not possible

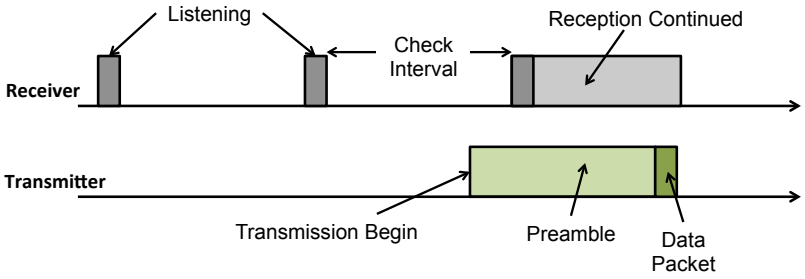


Figure 2.5: Low power listening mechanism used in the B-MAC protocol.

for a radio to transmit packets and listen for packets simultaneously in the same frequency band.

Different approaches have been designed and proposed for Medium-Access Control (MAC) so that several devices can share a common communication channel among themselves and achieve different performance trade-offs with respect to energy consumption, network throughput, and latency. A large number of sensor networking MAC protocols are inspired from Carrier-Sense Multiple Access (CSMA) protocols, where a transmitter listens to the channel (*carrier-sense*) before transmitting a packet. If it listens some activity on the channel implying another transmitter is already transmitting, it waits for sometime before attempting to transmit again. This helps in making sure that no two devices transmit at the same time. In sensor networks, medium access is typically used with radio duty-cycling so that the radio is not required to be kept on at all times.

One of the popular MAC solutions for sensor networks is called B-MAC [78], which is implemented based on the Low-Power Listening (LPL) concept as shown in Figure 2.5. A listener turns on its radio for a short duration after every check interval (usually 100 ms); if it hears any transmission on the channel then it continues to listen, otherwise turns off itself and listens again after the check interval. A transmitter should transmit a preamble for a time-length equal to or greater than the duration of the check-interval to ensure any listener in its radio-vicinity recieves its packet, which is transmitted immediately after the preamble.

Sensor-MAC (S-MAC) [79] is another popular CSMA-based protocol that saves energy by maintaining localized schedules and neighbors coordinate their listen and sleep cycles via exchange of SYNC messages. Collision avoidance is achieved by using CSMA-like approach. In WiseMAC [80], nodes listen to the radio channel periodically for short durations (a process called *channel-sampling*) in a fashion similar to the LPL mechanism. In contrast to B-MAC, WiseMAC lets an access point

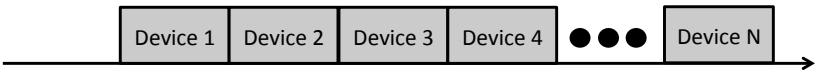


Figure 2.6: Time-division multiple access protocol

know the sampling schedule of all the nodes such that the access point starts the transmission at the right time and sends a significantly short preamble, thus saving energy. The sampling schedule of all devices is learned via acknowledgements received by the access points.

In contrast to CSMA, another approach to medium access can be based on Time-Division Multiple Access (TDMA) (briefly illustrated in Figure 2.6), where each device has a pre-decided slot to transmit, and the listener nodes know when to turn on their radios to listen to incoming packets. TDMA requires a complex setup phase in case of multi-hop topologies and may not be suitable for dynamic topologies. Moreover, there is a requirement of clock-synchronization among all the devices that causes additional overhead.

Traffic-Adaptive MAC Protocol (TRAMA) [81] is a TDMA-based algorithm that provides both random-access and scheduled-access periods. The devices are assigned slots in the scheduled access periods based on the neighborhood information within two-hop distance and the traffic in one-hop distance. The random access period is used for signaling such that the nodes can be synchronized and the slots can be assigned based on the neighborhood. Similarly, RT-Link [82] is another TDMA-based *multihop* protocol that instead of a contention/random access mechanism, uses an externally generated signal that acts as a global-beacon for time-synchronization. Each device has a special hardware that receives this signal and each device is assigned slots such that no overlapping transmissions occur. The motivation behind multihop networks and correspoding challenges are briefly described in the next section.

The Network-Harmonized Scheduling (NHS) protocol proposed in this dissertation is a TDMA-based protocol that not only handles medium access control, but also coordinates the packet transmission across a multihop network.

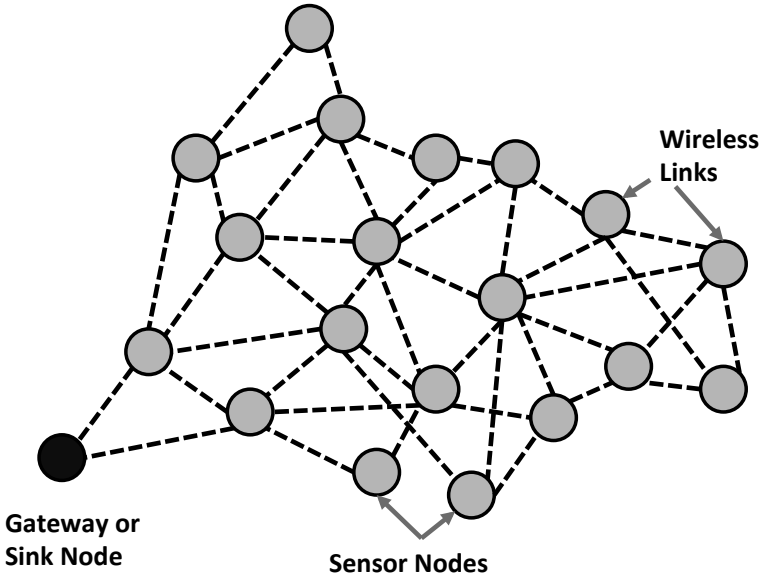


Figure 2.7: A symbolic representation of a sensor network where circular discs represent sensor nodes and dashed lines represent wireless links. The wireless links are the edges and nodes are the vertices of the network graph.

2.3.2 Routing and Data Collection

Medium Access protocols are responsible for the coordination between several devices when they attempt to use the wireless channel for transmission to ensure no two devices transmit simultaneously. Once the network is larger than what can be covered within the radio range of communication, multihop networking is required where data from a leaf node has to be delivered to a root node. Ensuring end-to-end connectivity while minimizing the energy consumption is one of the most important challenges in the domain of sensor networks.

In general, a source node can transmit the sensed data to the sink either directly via single-hop long-range wireless communication or indirectly via multihop short-range wireless communication. However, long-range wireless communication is costly in terms of both energy consumption and implementation complexity for sensor nodes. In contrast, multihop short-range communication can not only significantly reduce the energy consumption of sensor nodes, but also effectively reduces the signal propagation and channel fading effects inherent to long-range wireless communication, and is therefore preferred.

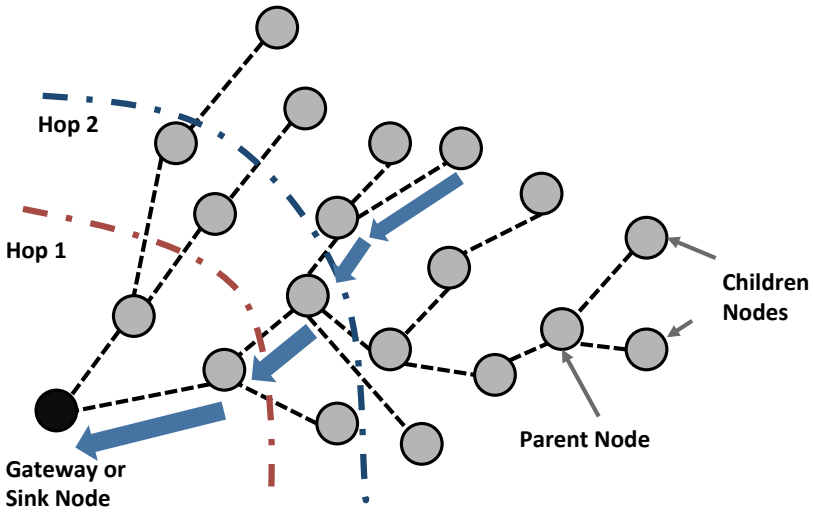


Figure 2.8: The network topology from Figure 2.7 converted to a spanning tree with the help of a routing protocol. The thick arrows show a data path from a leaf-node to the sink.

A sensor network is then organized in the form of a graph where each vertex corresponds to a sensor node, and each edge corresponds to a wireless link between two nodes. In a less generic form, a network graph can be organized in a directed or an undirected tree, where data can be collected at the sink node through various optimized paths or routes from all the nodes' paths to a sink. To illustrate the basic idea and terminology used in routing, we show an example sensor network in Figure 2.7, where sensor nodes are represented by circular discs and a wireless links by dashed straight lines indicating a bi-directional communication link. The network is in the form of a *graph*, and the overall structure of the network with its links is also referred to as *network-topology* or *topology*. A routing protocol can help reduce the network graph to a tree-like topology called a *spanning tree* in graph theory, rooted at the gateway or the sink node as shown in Figure 2.8, such that each node has one unique route to the gateway. Such a spanning tree allows any intermediate node to know the next node in the tree-hierarchy to forward the data to. The distance of a node from the sink in terms of number of edges is referred to as *hop-distance* or *hops*. A node at an i^{th} hop collects data from one or more nodes connected to it at the next $((i+1)^{th})$ hop. The former is referred to as a *parent*-node and the latter are their *children*. The nodes in a spanning tree without children are called *leaf-nodes*.

There are, however, several design issues that are unique to routing in sensor networks [83], and are outlined below:

Node Deployment Sensor nodes can typically be deployed in harsh climates and remote terrain. Even urban sensor deployments involving structural-health monitoring or industrial sensing can suffer from harsh operating conditions, where deployment is challenging. Nodes can be deployed randomly over a uniform field such as in agriculture monitoring or carefully placed at specific target points such as while monitoring the health of a bridge. In both such types of deployments, creating a optimal topology is a challenge. The routes have to be established such that all the nodes have connectivity.

Network Load Balancing While ensuring connectivity is important, it is possible that some nodes may have to handle more packets to forward than the others, thus leading to more energy consumption. If possible, the routing scheme is entrusted to find such bottleneck nodes and create new routes to ensure load balancing.

Fault Tolerance Wireless links can be intermittent, unreliable and asymmetric requiring networking protocols to provide redundant links or the transport layer to trigger retransmissions. Routing takes care of such issues depending on the application requirements such that varied degrees of fault-tolerance can be provided.

Scalability Routing protocols either establish static links at the start of the network or periodically refresh the routes or use flooding mechanisms where data is forwarded and re-forwarded to all the nodes in the network. In the case of new devices joining the network or an old device leaving, the network performance is different based on the type of routing used.

Data Aggregation If data needs to be collected at one or more sink nodes in the network, the number of packets to be forwarded by a node grows exponentially with the depth of the node in the network tree. The routing layer can make it possible to aggregate data through the network so that the number of packets do not accumulate at each hop. For example, if the application requires basic aggregates such as Max, Min, Sum or Average, then instead of sending all the raw data, each node can only forward a local aggregate based on the data received by it.

AdHoc On-Demand Vector (AODV) [84] and its variants are one of the most popular approaches to establish routes by finding shortest paths in the network graph. Such a routing protocol is then used for collection of data from all the nodes in the network at a root node. Resource-constrained sensor networks have severe energy constraints so protocols such as Trickle [85] adapt the rate of

transmission of updates to prevent incessant flooding in the network. For example, Collection-Tree Protocol (CTP) [86] finds routes using and AODV-like approach and then adapts them dynamically using a rate-control mechanism inspired from Trickle.

For implementing the Nano-CF programming framework, we used the BMAC-protocol with the AODV routing protocol to deliver applications to remote nodes and to collect data from them as well. Further details of the related protocols are provided in Chapter 3.

Summary

In this chapter, we provided a brief background on various technologies that are used in sensor networks in general. In particular, we stressed the ones we employed in the solutions proposed throughout this dissertation. We covered a wide range of technologies from hardware platforms to operating systems and networking protocols.

Chapter 3

Related Work

In order to acheive the goal of supporting and optimizing the execution of multiple applications on sensor networks, we have proposed a three-fold approach that includes a programming framework, an optimization scheme to eliminate redundancies and a network protocol to coordinate packet transmissions. A large number of projects has been undertaken in the past in these three broad domains. Hence, in this chapter, we summarize the existing research addressing the following areas while contrasting them with our proposed approach.

(a) programming of sensor networks,

(b) redundancy elimination, and

(c) networking protocols.

3.1 Programming Sensor Networks

Designing and deploying applications on a sensor network can be challenging because of the geographical spread of the sensor nodes, as well as the heterogenous nature of the hardware and software. In a traditional sense, the sensor networking applications are programmed at the operating system level, where a programmer may have to implement a distributed logic while having

working with very low-level details. This method makes it difficult to create large-scale applications quickly, especially for users from non-computer-engineering backgrounds. Researchers have long focused on developing programming tools that can help users from varied backgrounds to develop applications for sensor networks at a *macro* scale; this process of programming a set of nodes in a network is typically referred to as *macro-programming*. There is a wide spectrum of such macro-programming tools serving varied applications that range from programming frameworks designed to deploy application-specific virtual machines on sensor nodes to high-level programming abstractions that provide an abstract view of the network to a programmer. We will now discuss some of these programming tools classified into programming frameworks, middleware and programming abstractions.

3.1.1 Over-the-Air Programming Frameworks

Several programming frameworks have been proposed in the past that allow a user to deploy applications on sensor nodes over the wireless network rather than flashing a firmware by connecting a node individually to a computer. This method of programming sensor networks is referred to as *over-the-air* or *in-network* programming. Matè [32] is one of the earliest proposals that delivered an efficient virtual-machine to each sensor node running TinyOS, rather than sending an entire binary of a few kilobytes over the air. Virtual-machines in Matè are created using a special programming language called TinyScript such that a typical sensor networking application can fit within a few tens of bytes. A virtual machine is received and executed using a byte-code interpreter running on top of TinyOS instances on each sensor node. A simplified architecture of Matè is shown in Figure 3.1. An application created by a user is converted to byte-code, which is in turn split into one or more code-capsules of 24 bytes each, such that one capsule can fit in one packet. The capsules corresponding to an application are sent to sensor nodes over the wireless network and then they are stitched again to form the virtual machine. The Matè byte-code interpreter then executes the virtual-machine. The overall design and architecture of our proposed Nano-CF framework is similar to Matè, but in contrast, Nano-CF supports multiple applications and allows a programmer to specify a set of nodes that receive and execute a particular application.

Kairos [87] is a similar macro-programming approach that allows a programmer to write a high-level code specifying node interactions. It provides constructs like `get_available_nodes()` and `sleep()`

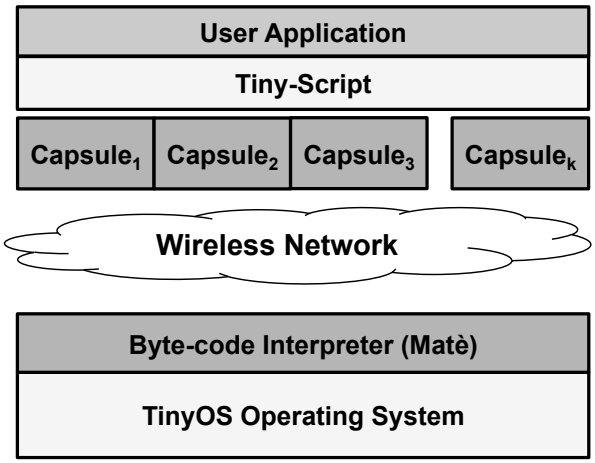


Figure 3.1: A simplified architecture diagram of the Matè virtual machine for sensor networks

to identify the presence of nodes in a network and control their active/sleep behavior, respectively. The high-level code is broken into node-level binaries that are delivered to all the nodes in the network.

More recently, solutions like PyoT [2] provide macro-programming support based on popular and standard protocols like 6LowPAN [9] and the Constrained Application Protocol (CoAP) [88]. Since it is built on top of well-known protocols, PyoT is suitable for rapid and interoperable application development. PyoT uses the Python scripting language as the shell scripting interface to create applications. It also provides a graphical web-interface for visualizing the network status and polling sensor values directly. The architecture of PyoT is shown in Figure 3.2. A PyoT worker node manages one sensor network and also acts as interface link between the sensor network and the Internet. The control center manages the overall network and provides the user with a view of the network through the web-interface.

The scheme conceptually closer and nearly contemporaneous to our Nano-CF programming framework is SenShare [3, 89] that is also designed to support multiple applications on a sensor networking infrastructure. SenShare supports multiple TinyOS applications on sensor nodes through a hardware abstraction layer that assembles low-level components into binaries to be executed on the underlying operating system. One of the major disadvantages of SenShare as compared to Nano-CF is that it has only been implemented for the Imote2 [90] sensor nodes running Linux operating system and it may be impractical for low-powered devices. Nano-CF, on the other hand, allows multiple applications

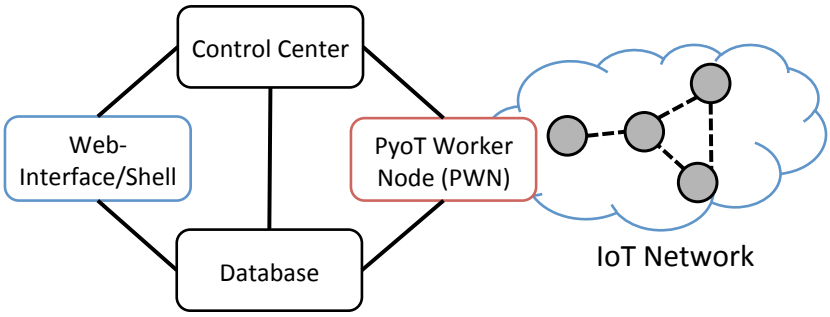


Figure 3.2: Architecture diagram of the PyoT macro-programming solution, as presented in [2]

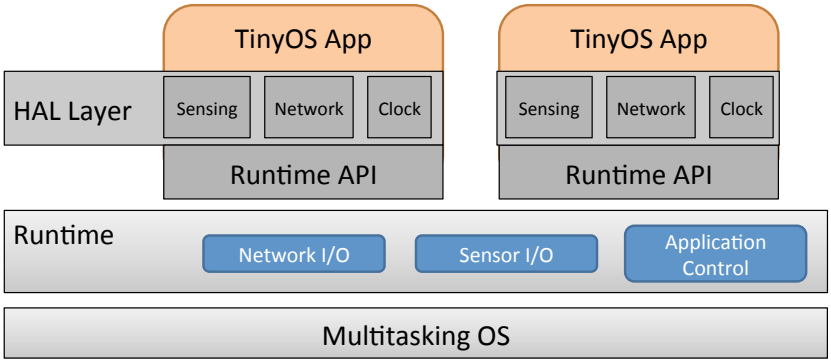


Figure 3.3: Architecture diagram of SenShare multi-application programming framework, as presented in [3]

to be deployed and supported on resource-constrained sensor networks, as recent versions of sensor node operating systems have support for multiple tasks [60, 35].

The process of reprogramming all sensor nodes incurs a large overhead as multiple application packets may need to be sent to individual nodes. Compressing the size of reprogramming packets [91] and incrementally programming each node [92] are some of the techniques proposed for reducing this overhead. The frequency of reprogramming the nodes is typically lower than that of data communication, hence it is more beneficial to optimize the resource consumption during the normal use of the network.

3.1.2 Middleware Approaches

Addressing the challenge of programming sensor nodes from a higher perspective, other works such as [93, 94, 95] describe middleware for facilitating the development of sensor networking applications on individual nodes. One of the most common middleware systems for generic computer systems, CORBA [96] hides the location of remote objects and hides the application's interactions with the remote objects. The Lime middleware [97], designed for mobile and ad-hoc networks, abstracts the underlying network and uses tuple-based interaction with the devices. Similar middleware have been designed for sensor networking, where network details are hidden. Often times, middleware-based approaches are limited in the flexibility of reprogramming, but act as an interface layer between sensor-networking primitives available to the applications and the underlying networking infrastructure. For example, Mires [94] is a message-passing middleware with a publish/subscribe model for collecting data from the sensor network; a high-level user application can select (subscribe to) a desired data that is published by the sensors. The lower level intricacies of connecting the sensors, establishing the routes and collecting the data are handled by the middleware, and are transparent to the user. This is similar to the DSWare approach [95] that allows a user-application to query the sensor network by creating an event in a high level programming language.

TinyLime [93], shown in Figure 3.4, is inspired from the Lime middleware, and is adapted to a more complex scenario where mobile base-stations collect data from a network of stationary sensor nodes. The users (clients) use a client host to connect to one or more base stations to exchange data in the form of Lime tuples. The base station then exchanges the information with the motes that are directly connected to it.

Query-based approaches for reprogramming sensor networks such as [33, 98] provide declarative programming expressions for processing data gathered by the sensor nodes. These schemes are useful to systems where different types of data are required frequently for monitoring and inference purposes. TinyDB [33] allows the use of SQL-like queries for getting aggregated responses from the sensor network in a transparent fashion. A TinyDB query consists of a clause of the type: `SELECT-FROM-WHERE-GROUPBY`. The functions of these clauses are similar to that of SQL. A user can specify a query in the following form:

```
SELECT nodeid, light, temp
```

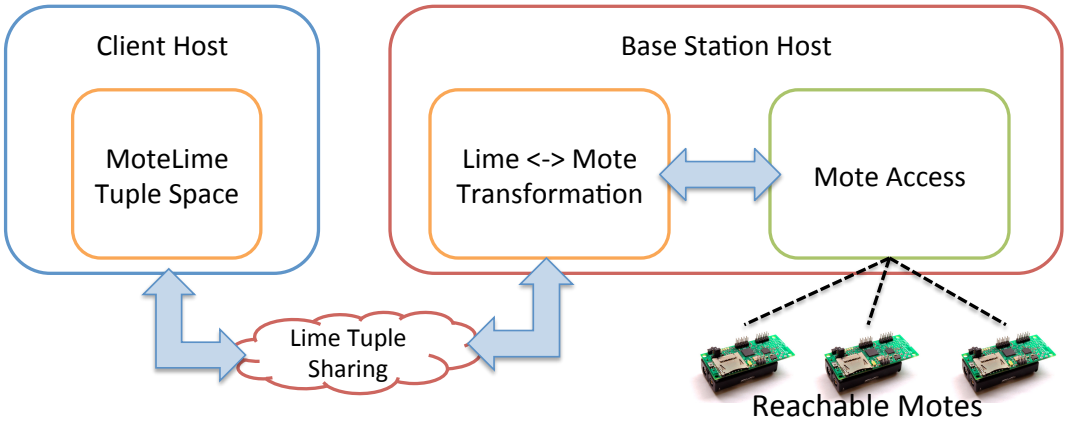


Figure 3.4: Architecture of the TinyLime middleware showing simplified interactions between a client, a mobile base station and the sensor motes.

```
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

TinyDB also has in-network aggregation mechanism to reduce the amount of data and number of packets that needs to be exchanged in the network. Such in-network query processing approaches are advantageous in applications where frequent processing of new but simple queries is required. These schemes are convenient to use, but not very scalable, as individual programming of every node may be required to implement a new query.

Most of the middleware solutions discussed so far, hide the network interactions from a user, and provide a convenient user-interface to access the required information from the network. Such schemes are, however, limited in the flexibility and are suitable mainly for data collection applications. They are not appropriate if the applications require more control (for example, changing the device interactions in the network). Moreoverer, access from multiple users can be implemented only in terms of sequential queries, which limits the possibility of efficiently using the underlying multi-tasking operating systems such as Contiki or Nano-RK.

3.1.3 Abstractions for Macro-programming

High-level programming abstractions like Regiment [31] and Hood [99] allow programmers to view the network as a set of abstract subnets based on neighborhood, proximity to an event, sensor reading or a combination of these. These abstractions allow convenient selection of nodes for reprogramming, data collection, and aggregation; thereby optimizing the overall communication in the network. Regiment is a macro-programming language and a runtime framework that presents the sensor network as a set of distributed, time-varying signals to a programmer. The signals can correspond either to the sensor readings, results of localized computation/signal processing or an aggregate value across a set or *region* of sensor nodes. Regions can be defined in terms of geographical area, network topology or capabilities (e.g., all the nodes with humidity sensors). Regiment uses a Functional Reactive Programming (FRP) model that includes a function `rmap(f,s)` to map a function `f` to a region of all sensors given by `s`.

On the other hand, the Hood programming abstraction allows a node to identify a subset of nodes around it based on a variety of criteria and share data with those nodes. Hood includes several programming interfaces that allow powerful-yet-simple interactions among devices to help in implementing large-scale and distributed applications. These interfaces allow setting of key properties of the network, such as attributes (e.g., sensor type), neighborhood size, read operation and write operation with other sensor nodes. Hood also incorporates a code-generation tool that generates the interfaces using the `generate` commands in nesC.

Logical Neighborhood [64] is also another powerful abstraction that replaces the conventional physical neighborhood defined based on communication range with a logical one, where the proximity of a node can defined by applicative information. Logical neighborhoods are specified declaratively using the specifically designed Spidey language. Our proposed framework, Nano-CF, can easily make use of such abstractions for subnet selection and also support multiple applications simultaneously over the whole network.

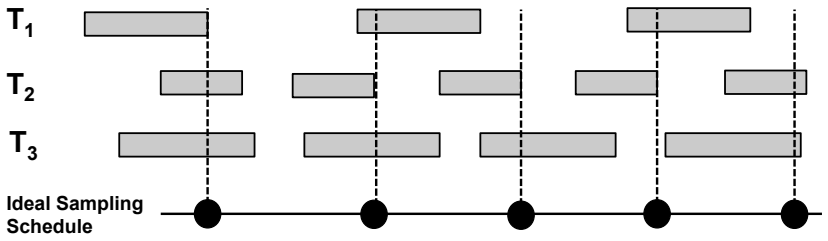


Figure 3.5: Finding the overlapping instants to collect samples across sensing windows of different tasks as proposed in [4]

3.2 Redundancy Elimination across Applications

Redundancy elimination is a common optimization strategy in compilers, but it is mostly limited to the case of a single program. Several compiler optimizations have also been designed for multiprocessor architectures for enhancing parallelism in sequential code [100, 101]. The direct application of such compiler techniques, however, is not possible in the case of sensor networks, because of the distributed nature of the network and the correlation of data to the physical environment and, hence, the physical location. A compiler for network-level programming of sensor networks should take into account the node characteristics including the hardware limitations and sensor peripherals, and the network interactions.

Recently, there has been interest in using sensing infrastructure for multiple concurrent applications. The scheme proposed in [4] describes a system called Task-Cruncher for sharing sensor readings among multiple tasks running on each sensor node by aligning sensing requests according to the periods, and sensing at time-instants providing the maximum overlap as shown in Figure 3.5. The system also reduces the communication incurred at a sensor node by combining data from each sensing task. The redundancy in computation is minimized by optimally merging the data-flow graph corresponding to each task. Such an optimization, however, is limited to an individual node. At a network scale, the interaction between the multiple tasks on multiple nodes, requires a higher-level framework for efficiently reducing the resource consumption in computation as well as communication. The solution proposed by the authors is a runtime algorithm that can significantly increase the scheduler and timing complexity on a sensor node. Moreover, that work is limited to finding overlap in case of one sensor per node, and efficiently extending it for multiple sensors is not trivial.

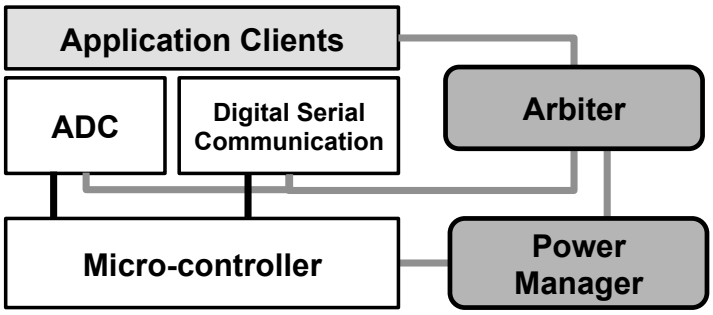


Figure 3.6: Simplified architecture diagram of the Integrated Concurrency and Energy Management (ICEM) approach, with Power Manager and Arbiter modules

Redundant sampling requests from different applications can also be handled at the device-driver level without major change in the application code. Integrating Concurrency and Energy Management (ICEM) [102] supports energy management at the device-driver level by providing explicit interfaces called power-locks, which applications can leverage to minimize the energy consumption. The concurrency and synchronization issues are transparently handled by ICEM. The power locks are managed by a module called Arbiter that manages the queueing policy, and interacts with the Power Manager as shown in Figure 3.6. In contrast, our approach of redundancy elimination across applications not only shares the data from external devices, but also simplifies the execution on a node by eliminating the complexities arising from a scheduler.

Furthermore, techniques for eliminating redundancies in sensor networks can also find inspiration from the field of database research, as several optimizations have been developed in the past to identify redundant queries. The approach of detection of common expressions proposed in [103] creates intermediate requests that assist reuse of intermediate data to save redundant accesses to overlapping sections of a relational database. Query optimization for detecting common data, as described in [104], also provides an improved solution based on interleaving smaller chunks of query execution. These schemes are limited to parallel or temporally close queries, and optimized for large data-sets. A window-based solution is proposed in [105] to share data among independent dynamically-issued queries. Similar schemes may be applied to reduce redundancies across multiple queries in query-based approaches for sensor networks (like [98, 33]) allowing temporal reuse of data-subsets. However, a node-level mechanism is still required to eliminate redundant sensing requests from different applications or queries.

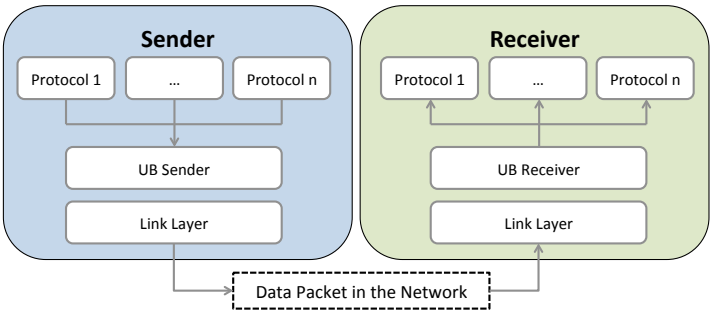


Figure 3.7: An overview of the Unified Broadcast approach as shown in [5]

3.3 Network Coordination

Reducing the radio-usage for data delivery in wireless sensor networks is a well-researched area. The solutions cover various aspects ranging from link-layer protocols to network flooding and distributed TDMA solutions. Network resource-consumption in case of multiple applications is a new challenge and not many works directly address this. The Low-Power Wireless Bus (LWB) approach [106] is a flooding mechanism where data from one or more *initiators* is flooded across the network. Data from each source is received by every other device, so it emulates a bus-like behavior even in the case of multi-hop networks. LWB is based on Glossy [107], a flooding and synchronization approach that leverages the property that multiple near-simultaneous and identical receptions at a node do not interfere, and these packets can be demodulated with a high degree of success. However, the flooding of packets, even if they are directed to a small subset of nodes, may lead to a high degree of redundant transmissions. Moreover, in the case of multiple applications, the resulting overhead can become prohibitively large.

In addition, there are many approaches that aim at reducing the amount of time that a node has the radio in the ON state by defining a periodic wake-up scheme. Typically, these approaches are distinguished between synchronous and asynchronous. In synchronous approaches, nodes agree on a common sleep/wakeup schedule [108, 109] to save energy. However, in such schemes, nodes may be forced to maintain several sleep/wakeup schedules if their neighbors have more than one schedule. Asynchronous approaches are based on channel polling [78, 110]. In these protocols, nodes periodically wake up and try to sense the channel, and if the channel is active, then nodes stay awake to receive transmissions.

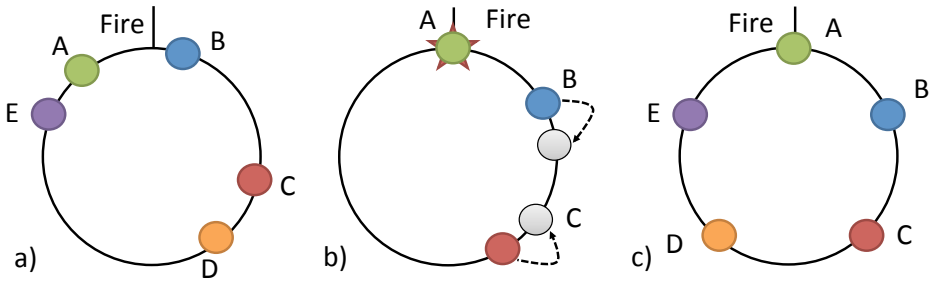


Figure 3.8: Working of the DeSync Algorithm as illustrated in [6]: a) Random transmission schedule, b) Nodes B and C adjust their transmission after listening to A's transmission, c) Stable desynchronized state.

Similar to our goal of batching data from multiple applications, the Unified Broadcast [5] approach transmits unified data from various services running on a sensor node as shown in Figure 3.7. In Unified Broadcast, data from various services is transmitted when the number of accumulated packets reaches a certain threshold. This approach is limited to sending data together from multiple services, and therefore it is only valid for broadcast messages. In the case of multiple applications requiring many-to-one communication, an additional protocol is required. The work by Hansen *et al.* [111] has shown experimentally that it is still possible to preserve correctness of a set of representative WSN protocols (such as FTSP [112], Trickle [85] or CTP [86]) when packets are delayed to minimize network resource usage. In the case of Unified Broadcast, the periods of the applications are implicitly detected when a second packet of the same protocol is requested to be transmitted. As we will see, in this work, we explicitly take the period of the application into account to derive the harmonizing period of communication.

The extreme of sleep/wakeup schemes is Time Division Multiple Access (TDMA), where nodes only wakeup at the scheduled transmit/receive times, at a cost of tight synchronization and no flexibility to changes. An interesting perspective to minimize the probability of collision such that nodes transmit independently in non-overlapping time slots is proposed in Desync [6]. The Desync approach forms a round-robin TDMA schedule for reducing the power consumption. With the Desync protocol, nodes are receptive to the neighbor node's firing (transmission) period and they adjust their offset equidistant from each other in a cyclic way. This mechanism is illustrated in Figure 3.8, where starting with a random schedule of transmission of 5 nodes, the firing pattern*desynchronizes* over time, implying that all the nodes are at evenly spaced in time. This scheme, however, is limited to single

broadcast domain, and it is not easily extendable to multi-hop communication scenarios. TDMA for multi-hop is typically achieved using 2-distance graph-coloring algorithms. Such protocols require much more information about the network topology, and typically a central coordinator, as it is done in RT-Link [82] or Distributed TDMA [113]. In contrast, our approach aims to achieve TDMA-like efficiency without global state maintenance.

Summary

In this chapter, we outlined the related work addressing the challenges in programming sensor networks, eliminating redundancies and network coordination. We argued that supporting multiple applications on a sensor network infrastructure is a relatively nascent research area. A few research studies have addressed this issue but they target hardware and software platforms that differ significantly from those covered in this dissertation. Furthermore, optimizing the overall resource-consumption in sensor networks with multiple applications brings several other challenges that need to be addressed from a different perspective.

In the next chapter, we describe our proposed Nano-CF programming framework that supports multiple applications on resource-constrained sensor nodes.

Chapter 4

The Nano-CF Programming Framework

Wireless Sensor Networks (WSN) are increasingly being deployed for large-scale sensing applications such as building monitoring, industrial sensing, and infrastructure monitoring. Often, sensor networks are deployed in difficult terrains and it is expensive to re-program all the nodes individually, seriously limiting the manageability and usability of sensing infrastructure. Several macro-programming schemes [32, 31, 98, 87, 114] have been proposed in the past to abstract away from low-level details of sensor networking such as radio communication, analog-to-digital converter (ADC) configuration, memory management and reliable packet delivery. Macro-programming systems typically provide a unified high-level view of the network, allowing programmers to focus on the semantics of the applications to be developed rather than understanding the diverse characteristics of underlying platforms.

A framework, which supports multiple users to write independent applications and execute them seamlessly over a given sensor networking infrastructure, can be highly beneficial for sensor network researchers and other interested users. Such a system should allow the users[1] to use the sensor network without being concerned about other users' applications on the same network.

[1]We use the terms 'users' and 'programmers' interchangeably in this chapter, as the proposed solution is designed to support the users who are interested in programming the sensor network.

4.1 Motivation

Many real-world deployments suffer from the problems of limited usability and low involvement of users, either because *(a)* the sensors are expensive and it may not be practical to deploy them with ideal or desired density, or *(b)* middleware support to allow seamless deployment of applications is inadequate. The former is addressed in [115] by sharing sensors among multiple deployments through human involvement and to address the latter, design requirements for a middleware to support concurrent applications are outlined in [116]. A typical use-case of supporting multiple applications simultaneously can be conceptualized on a university test-bed deployment like Sensor Andrew [29]. Sensor Andrew is a sense-actuate infrastructure deployed across the Carnegie Mellon University campus. The test-bed is used for inter-disciplinary research ranging from link-layer protocol development to design and testing of applications such as building-energy estimation and social-networking support systems like neighbor discovery.

All the nodes in Sensor Andrew need to be programmed individually for supporting any new application. Furthermore, a user should contact a system administrator to re-program the network. As this test-bed is an interdisciplinary effort, researchers from different technical backgrounds use the infrastructure for their needs. To help better understand the goal of a middleware framework on a sensor deployment, the following simple example can be used. Consider a task that monitors temperature and humidity in various buildings regularly, and reports it to a civil engineering researcher interested in constructing an air-flow map of the building. In another task, a building manager might be interested in using the same infrastructure for a high-priority deadline-based fire alarm system. Users from such diverse technical backgrounds may benefit from a middleware layer that helps them program the network at an abstract higher level. A conventional macro-programming framework, however, may not allow more than one user to independently program the sensor network for handling these kinds of applications simultaneously. Furthermore, supporting multiple applications pose the additional challenge of coordinating tasks on every sensor node and scheduling radio transmissions. We propose a framework called *Nano Coordination Framework (Nano-CF)*, to provide support to multiple programmers to write independent applications for a given sensing infrastructure. The framework seamlessly deploys those applications on the end nodes, and coordinates the packet delivery and data aggregation to reduce overall resource usage in the network.

The proposed framework has a global view of various applications running on the network and

their mutual interactions, it batches the sensing tasks and radio-usage together, with the help of Rate-Harmonized Scheduling (RHS) [41]. RHS proposes a scheduling scheme to maximize the sleep duration of a processor in case of periodic tasks. We adapt RHS to coordinate periodic radio usage tasks such that the packets from several tasks can be transmitted together and smaller packets can combined into larger ones . It is shown in the subsequent sections that significant savings in processor use and packet transmissions can be achieved through such a batching mechanism. In Nano-CF, multiple tasks are coordinated based on their timing parameters, within an allowable deviation as specified by the user. The major contributions of our proposed coordination framework are as follows:

1. It facilitates the use of a sensor-networking infrastructure by multiple programmers for multiple independent applications simultaneously.
2. It leverages the real-time and resource-centric features of an underlying sensor-network operating system for providing low-latency response.
3. It also improves the network lifetime by clustering processor usage and radio communication.

4.2 System Design

Nano-CF (Nano Coordination Framework) is an architecture for macro-programming with coordinated operations in a WSN that encompasses multiple layers of a sensor networking system architecture, as shown in Figure 4.1. A main consideration behind the design of our framework is to support data-collection applications. Complex applications with actuation and distributed decisions can still be supported. However, the complexity of the resulting program can be quite high as the framework is not optimized for such programs.

A user interacts only with the top layer of our system, which we call the *Coordinated Programming Environment* (*CPE*). For developing applications on the sensor network, a user only needs to write programs using the Nano-Coordination Language (Nano-CL) we developed for Nano-CF. The remaining functionality of providing the abstraction from the lower-layer networking and topology is handled by the CPE. The functions of the framework are divided into two main aspects: *(i)* to handle *control* information including the re-programming packets, and *(ii)* to collect *data* from sensors. All the three layers contribute towards the exchange of control information and data gathering. The CPE

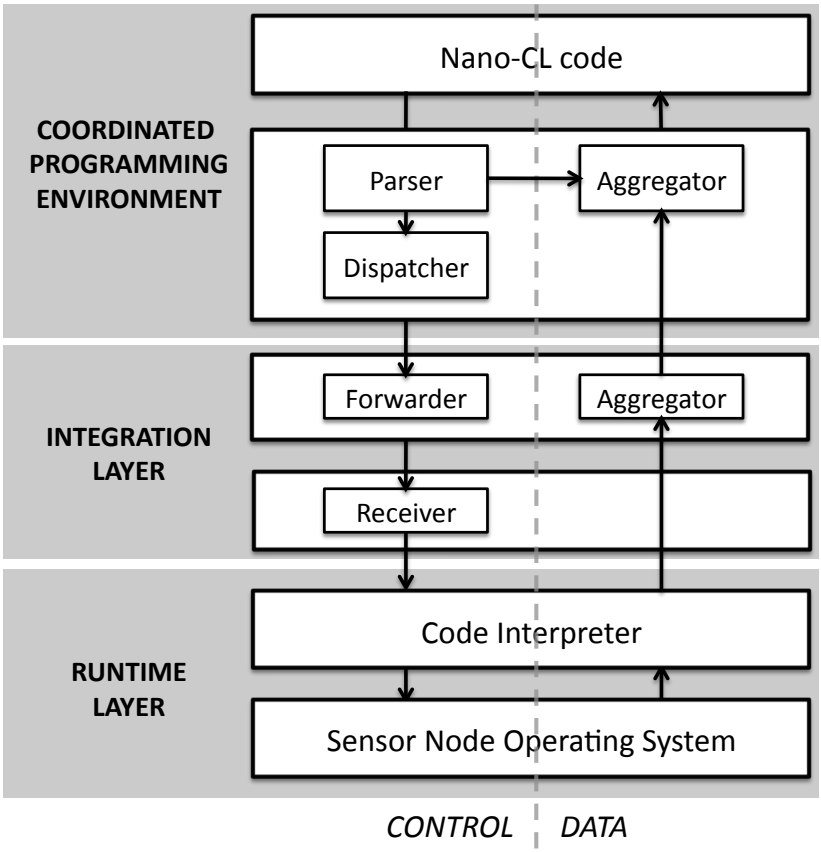


Figure 4.1: Layered architecture of Nano-CF

provides a programming interface to allow the user to write, compile and send programs over the air. The CPE also returns the aggregated data corresponding to each the application independently to the user. The CPE consists of a parser and a dispatcher. The parser or *compiler* converts the functional definitions specified by the user to lower-level byte-codes and then the dispatcher sends them to the deployed sensor nodes. We assume that multiple WSNs are connected to each other via the Internet, where each WSN is composed of sensor nodes with at least one gateway node. The communication from the CPE to the end nodes is handled through the *Integration Layer* (*IL*).

The Integration Layer encompasses all the nodes in the network and interfaces to the CPE through a gateway node. The gateway node implements a forwarder function to associate a specific task to a particular node and send corresponding programming packets to end nodes. An Aggregator module spreads over all three layers in Nano-CF to gather data from children nodes at a parent node in the

subnets and finally present the result to the user interacting at the CPE layer. In addition to packet delivery and data aggregation, it is the function of the Integrator Layer to make sure that the timing properties of the applications specified by the user are delivered to the OS. THE Integration layer also handles the responsibility of batching tasks and packet transmissions together using RHS, details of which are provided in Section 4.4.3.

The *Runtime Environment* of Nano-CF is implemented at each sensor node, above the operating system. At the lowest layer of this architecture, each node runs a byte-code interpreter to translate low-level instructions from the dispatcher to an executable form for the sensor node. Our architecture is highly portable because we only need to change the code interpreter to support different sensor network operating systems. Further implementation details along with the source code are provided at the project website [117].

4.2.1 Nano-RK: a Resource-centric RTOS for Sensor Nodes

Nano-RK [35] is a Real-Time Operating System (RTOS) designed and implemented to support resource reservations for wireless sensor nodes with a multi-hop packet transmission. By leveraging the timing characteristics of Nano-RK, such applications can be easily offered through Nano-CF. Virtual energy reservations introduced in Nano-RK also help Nano-CF to manage energy consumptions in each sensor node. By setting reservation values of *(CPU, Network, Sensor)* for runtime environment in a sensor node, we can enforce the expected power consumption. Since most other popular sensor node operating systems do not support multi-tasking by design, our current implementation is limited to Nano-RK. We aim to support more operating systems as future extensions of this framework.

We used FireFly [1] sensor nodes with the Nano-RK operating system for our framework. Each node has an Atmel ATmega1281 processor and a Texas Instruments CC2420 Transceiver for IEEE 802.15.4 compliant wireless communication. In addition, a custom sensor expansion card can be connected to the FireFly main board. In particular, the sensor expansion card offers voltage sensing, dual axis acceleration, passive infrared motion, audio, temperature, and light. These diverse sensors supported by the FireFly platform suit the goal of Nano-CF to deploy multiple applicatons simultaneously.

4.2.2 Routing and Link Layer

Since Nano-CF requires byte-codes to be transferred to each sensor node in a multi-hop networking environment, the framework requires the support of underlying routing and link layer protocols. For the purpose of brevity, the details of the routing and MAC layer are not discussed in this chapter, and a brief background of related technologies is provided in Chapter 2. Our framework is flexible to operate over any transport layer as long as the gateway node is able to communicate with every node through unique addresses. Many sensor networks employ modified versions of routing protocols such as AODV [84] and DSR [118]. We used a DSR-based routing protocol with multicast for our evaluation. Below the routing layer, the link layer is crucial for data delivery. Time-Division Multiple Access-based RT-Link [82] and contention-based B-MAC [78] are two commonly used protocols with Nano-RK. In our implementation, we opted for B-MAC, but the operation of Nano-CF is independent of the lower-layer protocol used.

Based on this fundamental architecture, we now describe the three main components of Nano-CF; Nano-CL in Section 4.3, the Integration Layer in Section 4.4, and the Runtime Layer in Section 4.5.

4.3 Nano-CL

We designed an imperative-style language called Nano-CL (Nano Coordination Language) that provides a unified interface to users for writing sensor networking applications. The language has been designed to meet the following design goals:

1. The language should provide an abstraction from the lower-level details of the sensor networking OS and radio communication.
2. The language design should facilitate the extraction of timing and communication properties from user-written applications.
3. The syntax of the language should be simple and easy for non computer-scientists to understand and program.

Each Nano-CL program is composed of two important sections: Job descriptor and Service descriptor, as shown in Figure 4.2.

```
 JOB:
<service1> <nodes> <rate> <agg_func>
<service2> <nodes> <rate> <agg_func>
  . . .
 END

 SERVICE:
<service1> <return_type>
/* instruction 1 */
/* instruction 2 */
  . . .
 END
```

Figure 4.2: A figure showing the format of a Nano-CL program, consisting of a job descriptor and a service descriptor.

4.3.1 Service Descriptor

In Nano-CL, the user writes a service which is functionally equivalent to a task that is to be executed on each node. Nano-CL consists of a set of primitives and programming constructs which provide sufficient capability for programming the sensor nodes, as well as, an abstraction from the lower-level implementation details of the operating system and radio communication. Each service descriptor specifies the functionality of one task. The syntax for a service descriptor is similar to 'C'-like sequential programming, where the user can make use of pre-defined library functions to interact with the sensor hardware. The return value from the service corresponds to the data value that the user wishes to collect from the sensor nodes, and, unlike the usual practice, more than one data value can be returned. The framework converts the user program in service descriptor into byte-codes, which are then sent over the wireless network to be interpreted and executed at each node.

4.3.2 Job Descriptor

A programmer can write multiple services and then each service can be mapped to a set of nodes in the job descriptor. The job descriptor section can have more than one service call where each call has the associated timing properties specified by the user. The timing properties include the periodic rate at which the service should repeat at each node and the maximum allowable deviation from the specified period. This deviation allows the framework to "batch" tasks together on sensor nodes,

as well as, schedule the transmissions together to reduce the overhead associated with switching on/off the radio and processor. This coordination of tasks and packet delivery across the network is explained in Section 4.4. A set of nodes given by `<nodes>` in the Job Descriptor section contains a list of all the nodes where the respective service should be executed. All the nodes in the network are assumed to have a unique identity, and are also mapped to a physical location in the network. The choice of having an explicit node-list to map the service is deliberate as Nano-CF can leverage the adaptive selection of nodes using some of the techniques already proposed in the literature such as [99, 119]. The routing layer can provide information about the network topology regularly, which can be used to dynamically select nodes in the job descriptor. The role of the `<agg_func>` is explained in more detail in the next section.

4.3.3 Nano-CL Compiler

The Nano-CL compiler (nclC) converts the source code consisting of services into byte-code streams. The compiler is designed with an aim to limit the byte-codes to a small subset of op-codes to allow the code-interpreter task on the end-node to have a small memory footprint. nclC adds metadata to the byte-code stream which helps the integration layer to extract information for batching the computation and radio usage on each node. It also specifies the timing properties for network-wide packet clustering. The metadata in the byte-code stream are generated from the information provided by the user in the `rate` section of the job descriptor, which consists of the period of the task and the allowable deviation from the period.

The following are the timing parameters handled by the compiler:

- `T_srv:` Repeat rate of the current service.

- `Dev_srv:` Allowable deviation in the repeat rate of the service.

The metadata are then sent along with the byte-code to individual nodes and are interpreted at the integration layer and the code interpreter.

4.3.4 Example Nano-CL Program

We provide a simple example of a Nano-CF program with two applications implemented using two services, shown in Figure 4.3. The aim of the first application is to find the number of occupied rooms in a building. We use a small network of four nodes at locations `L1,L2,L3,L4` with one node in each room. We assume that each of the nodes has light, audio and temperature sensors, and are placed in such a way that the occupancy of the room can be determined by one node. Service `occup_monitor` returns value `1` if room is occupied or `0` otherwise. Occupancy[2], in this example, is determined by comparing an average of 10 samples of a weighted combination of light and audio levels in the room against a threshold. Various parameters in this example are chosen based on experimentation, and may not be applicable in all cases. In order to determine the number of occupied rooms from these locations of sensors, we use the `SUM` aggregation function. In second service `temp_collect`, the user just wishes to collect readings of temperature every 50 secs. Please note that each of the services in this example could be created and programmed to a given sensor infrastructure by multiple users independently using Nano-CF. We have shown these services in a single program for ease of presentation.

4.4 Integration Layer

The Integration Layer (IL) is responsible for byte-codes delivery, data aggregation, optimization of task execution and data transmission on the nodes in the network. This layer, shown in Figure 4.4, overlays across the gateway and the endnodes.

4.4.1 Byte-code Delivery

The Forwarder module on the gateway node forwards the byte-codes generated by Nano-CL to the Receiver module on the end nodes. The primary features related to byte-code transfer are routing table management and fault-tolerant packet delivery.

The routing table generated during the network initiation phase is used for communicating with the end nodes. Sequence numbers, end-to-end acknowledgements and packet retransmissions must

[2]We are providing a simple example for understanding purposes. Precision of monitoring is not a goal of this example.

```
 JOB:
occup_monitor <L1,L2,L3,L4> <20s,5s> SUM
temp_collect <L1,L2,L3,L4> <50s,0s> NOAGG
 END

 SERVICE:
occup_monitor uint8
int16 light_sense, audio_level;
int32 sum;
int8 cnt, thresh;
sum = 0;
cnt = 10;
thresh=40;
for(i=1:cnt)
    light_sense = gets(LIGHT);
    audio_level = gets(AUDIO);
    sum = sum + light_sense/100;
    sum = sum + audio_level/100;
    wait(1s);
endfor
if(sum/cnt > thresh)
    return 1; // Return 1, if room is occupied
else
    return 0;
endif
  END

 SERVICE:
temp_collect uint16
return gets(TEMP);
 END
```

Figure 4.3: An example Nano-CL program with two services

be used because one missing packet may cause the network to malfunction due to missing instructions. When end nodes try to reply to a request from their gateway, a broadcast storm problem may occur. In order to avoid this problem, packets are scheduled with an offset as explained in the next section. Each packet header contains additional fields like packet type, source address, destination address, application identifier, re-programming packet sequence number, packet identifier and total number of packets for this application.

4.4.2 Data Aggregation

The Integration Layer covers gateway node and end nodes, and links the programming environment to the runtime layer on end nodes. Nano-CF supports in-network data aggregation through the Aggregator module for reducing the packet overhead in the network. The aggregation scheme is defined by the user in the Job Descriptor of the program. In our current implementation, we support common commutative aggregation functions given in Table 4.1. The Aggregator handles different

Table 4.1: Aggregation functions supported in Nano-CF

Function	Description
`MIN`	Minimum value of data
`MAX`	Maximum value of data
`SUM`	Sum of all data
`COUNT`	Number of replies received
`NOAGG`	Forward all data to CPE

functionality at different levels. At the leaf nodes, the job of the aggregator is to send its own data. At at an intermediate level in the network, it should combine its data with that from all the child nodes according to the specified aggregate function. If the function is `NOAGG`, then the intermediate node concatenates data from each of the child nodes along with the node id and forwards it towards the gateway. The aggregator module at the gateway node communicates directly with the CPE and provides the aggregated data to the user. Whenever the Receiver module receives a new re-programming packet from the gateway node, it coordinates with the runtime layer to manage multiple applications from various users.

4.4.3 CPU/Data Coordination

The IL provides task/network coordination for saving energy on each sensor node. Because most WSN applications are periodic, we utilize Rate-Harmonized Scheduling (RHS) [41].

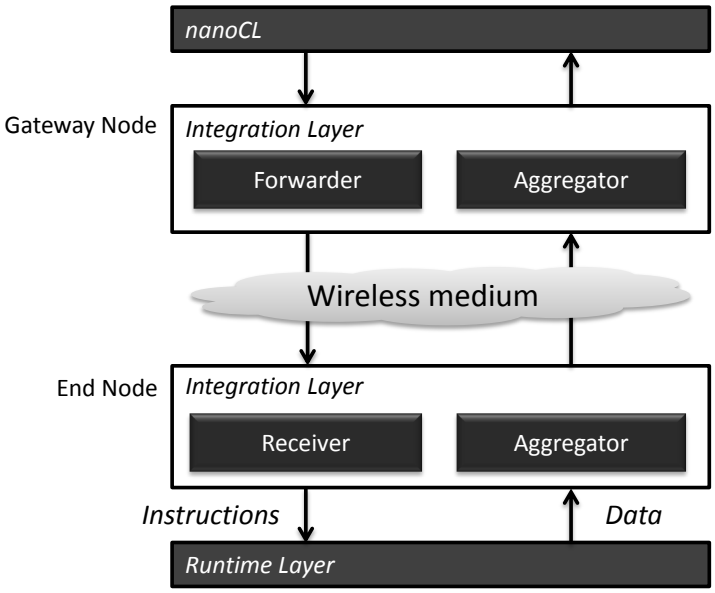


Figure 4.4: Architecture design of the Integration Layer of the Nano-CF programming framework and its interactions with the other components in the framework.

Notation and Assumptions

Suppose that each sensor node n_i is running a set of tasks, Γ_i, which is composed of m tasks, $\tau_1, \tau_2, ..., \tau_m$. Each task, τ_j, is represented by (C_j, T_j, D_j), where C_j is its worst-case execution time, T_j is its period, and D_j is its deadline. Tasks are arranged in a non-decreasing order of periods. The response time of τ_j is denoted as R_j. We assume that T_j is equal to D_j. Each task τ_j may also generate a B_j-byte packet ρ_j every $P_j \geq T_j$ time units. Thus, ρ_j can be represented as (B_j, P_j). A packet ρ_j is not dropped at the task level even if it may be lost in routing or the link layer. With P_j being the relative deadline for sending the packet ρ_j, a separate communication task in each sensor node is used to send these packets.

RHS clusters periodic tasks such that all task executions are grouped together in time to accumulate idle durations in the processor schedule. This accumulation helps each processor to get a chance to go into a deep sleep state. This property is also applicable to packet transmissions, and sending bigger concatenated packets will consume less energy than sending multiple packets more frequently.

Composition with Rate-Harmonized Scheduling

Let packet response time be the time duration from sensing the environment to the instant when the packet is delivered. Then, RP_j denotes the packet response time of ρ_j. The harmonizing period of tasks, $\mathbf{T_H}$, is chosen so that $\mathbf{T_H} = T_1$ if $\Psi = \emptyset$ and $\mathbf{T_H} = \frac{T_1}{2}$ if $\Psi \neq \emptyset$, where $\Psi = \{\tau_j | T_j < 2T_1, j \neq 1\}$, and $T_j < T_i$ satisfies if $j < i$.

Now we prove some properties of RHS with data clustering.

Lemma 4.4.1. *If a packet is generated by every job of τ_j, the worst-case packet response time, RP_j, for any packet ρ_j is bounded by $2T_j$.*

Proof. In the worst case, sensing the environment data can occur at the start of task execution, and packet delivery can occur at the end of task execution. Therefore, RP_j can be represented as $R_j + T_H - \epsilon$, where T_H is added because a packet delivery can be delayed for $T_H - \epsilon$ if a communication task just misses the harmonization boundary. If ϵ is infinitesimal and T_H is T_1 as the worst case compared to $\frac{T_1}{2}$, RP_j is $R_j + T_1$. Since R_j is bounded by T_j due to the implicit deadline of τ_j, $RP_j \leq T_j + T_1 \leq 2T_j$. □

Corollary 4.4.2. *Any packet, ρ_j, generated by τ_j will meet its packet transmission deadline if $P_j \geq 2T_j$.*

Proof. By Lemma 4.4.1, if $P_j \geq 2T_j$, ρ_j will be delivered within P_j. □

Theorem 4.4.3. *If τ_c is the communication task and represented by (C_c, T_c), a set of given tasks, Γ, is schedulable if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_c}{T_c} \leq \frac{1}{4} \tag{4.1}$$

Proof. This follows from *Lemma* 4.4.2 and *Theorem* 4[‡] from [41]. □

The result from *Theorem* 4.4.3 can be pessimistic because we strictly applied the packet deadline. If $P_j \gg T_j$ for τ_j and ρ_j, Equation (4.1) can be changed into $\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_c}{T_c} \leq \frac{1}{2}$, which is the same as the result from [41].

[‡] Theorem 4 from [41] proves that a taskset is feasible under basic rate-harmonized scheduling if $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 0.5$.

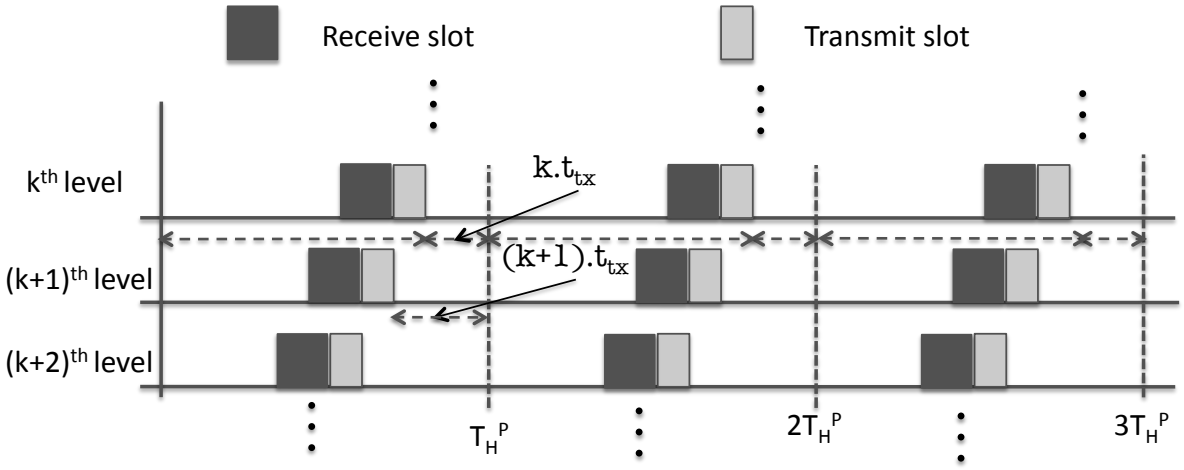


Figure 4.5: A time-line showing the possibility of network-wide batching at various hops in the network that can be achieved based on the Rate-Harmonized approach.

Energy-Saving with RHS

By using RHS for tasks, a processor in a sensor node can go into a deep-sleep state more frequently (at $\mathbf{T_H}$ boundaries). Applying RHS to packet transmissions allows each sensor node to send a merged packet instead of sending packets whenever it has data in the queue. The amount of energy saved by using RHS for tasks can be obtained from the length of the deep-sleep period given in the form of (C_{sleep}, T_{sleep}) [41]. The amount of saved energy can be derived from the number of transmitted packets. The number of transmitted packets per unit time when we do not use RHS is given by $\sum_{i=1}^{n} \frac{1}{T_i} \lceil \frac{B_i}{B_{max}} \rceil$, and by $\sum_{i=1}^{n} \frac{1}{\mathbf{T_H}} \lfloor \frac{T_i}{\mathbf{T_H}} \rfloor \cdot \lceil \frac{B_i}{B_{max}} \rceil$ when we use RHS. Here, B_{max} is the maximum packet size. We will evaluate these energy savings later.

4.4.4 Network-wide batching using RHS

As has been emphasized in earlier sections, a network programming framework like Nano-CF provides a global view of the network where efficient scheduling for packet aggregation at multiple hops becomes feasible. An efficient approach to aggregate packets and schedule data has been proposed in [120], however, we use Rate-Harmonized Scheduling across a network to save energy by reducing the frequent turning On or Off of the radio. The main reason to use RHS is that it can help batch

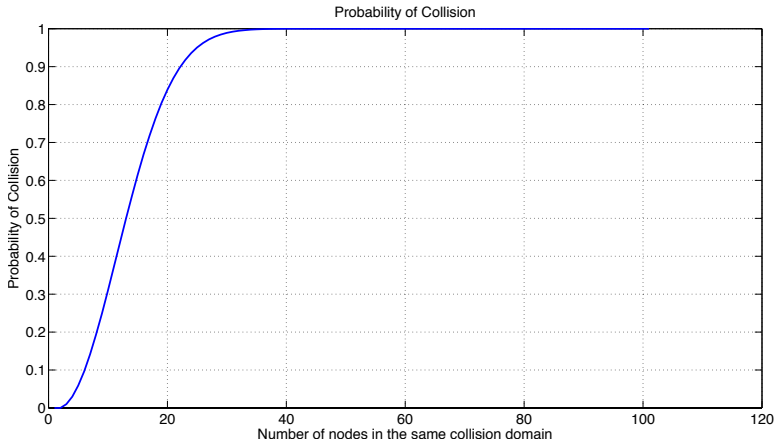


Figure 4.6: Probability of collision in a network when the nodes present in the network can transmit uniformly at anytime within the period.

tasks together even if the periods of the various network transmissions mismatch. We use RHS in a distributed fashion to batch packet events to be scheduled at the end of the harmonizing period T_H. However, if all the nodes in a subnet are scheduled to transmit at T_H, multi-hop aggregation of the packets cannot be supported, and packet collision will be high. However, if the packet transmission at the k^{th} hop can be offset to an earlier time, a parent node in the network can efficiently collect data from its children.

For efficient data collection, it can be deduced that each node should transmit at an offset given by:

$$\Omega_k = -(k \times t_{tx}) \tag{4.2}$$

where, Ω_k is the introduced offset of transmission from T_H, t_{tx} is the maximum amount of time a node uses its radio while transmission and k is the depth of the node in the tree. Equation 4.2 gives a simple schedule for packet transmissions in a multi-hop scenario. The nodes listen before they transmit as shown in Figure 4.5 and this schedule can be maintained with some coarse-grained time synchronization even over CSMA (Carrier Sense Medium Access) protocols. This allows collision free scheduling in the network, whereas if the nodes can transmit uniformly anytime within the T_H duration; the probability of collision of any two packets with n nodes is given by:

$$P_c = \frac{N!}{(N-n)! \times N^n} \tag{4.3}$$

where N is the number of possible slots in the harmonizing period, and is given by T_H/t_{tx}. Figure 4.6 shows the variation of P_c with respect to n for $N = 100$. It can be seen from the plot that the probability of collision quickly increases with the increase in number of nodes.

In this section, we briefly described an approach that coordinates packet transmissions in a multi-hop network to reduce energy-consumption and collisions. Further details of this approach are provided in Chapter 7, where this coordinating mechanism is included in the proposed Network-Harmonized Scheduling approach.

4.5 Runtime Architecture

The runtime layer of the framework consists of routing, communication and execution of byte-code on individual sensor nodes. In our current implementation, each sensor node in Nano-CF uses Nano-RK. The runtime environment has three types of tasks running on the OS: Receive (RX) Task, Transmit (TX) Task, and a set of code interpreter tasks. There are pre-defined copies of the code-interpreter tasks on each sensor node, corresponding to the number tasks to be supported in the framework. The RX and TX tasks take care of reliable packet delivery and also implement the routing layer.

4.5.1 Routing

The framework requires at least a basic routing layer to ensure connectivity to all the nodes. In our current implementation, we use a routing protocol similar to Dynamic Source Routing (DSR) [118], but the system is flexible with respect to the routing layer as long as the higher layer is able to address a node directly. The system can also support on-demand routing schemes, provided the user generates a topology-map of the network before reprogramming the network nodes. The topology map can be generated during the initiation phase and is beyond the scope of this manuscript. The user can also make use of `send()` and `receive()` primitives available in the language for developing ad-hoc routing schemes.

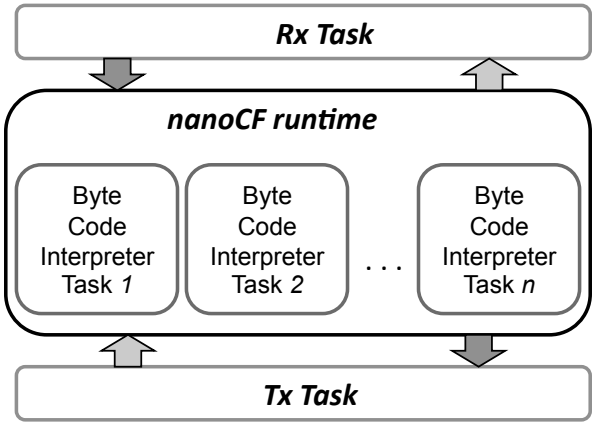


Figure 4.7: Block diagram of the runtime layer of Nano-CF

4.5.2 Code Interpreter

As shown in Figure 4.7 the code interpreter receives byte-code from the Nano-CL compiler through the *RX Task*. Table 4.2 lists the primary opcodes handled by the code interpreter. First, the code interpreter reads the metadata section of the byte-code and it saves the period (repeat rate) of the service `T_srv` and the deviation `Dev_srv` into local variables. The interpreter has a local instruction stack and it executes the byte-code corresponding to each instruction in a sequential manner. The interpreter maintains a local stack of variables and a return stack to support function calls. It repeatedly executes the byte-code with a period of `T_srv`, and also listens for any new byte-code packets for node reprogramming. If the code-interpreter task receives a signal from the *RX task* that it has received a new service to execute, it finishes the execution cycle of the current task, flushes the local stacks and copies the new metadata and instructions into the local memory and restarts the execution.

We now propose a programming pattern named *sMapReduce*, inspired by the Google MapReduce framework, for mapping application behaviors on to a sensor network and enabling complex data aggregation. The proposed pattern requires a user to create a network-level application in two functions: *sMap* and *Reduce*, in order to abstract away from the low-level details without sacrificing the control to develop complex logic. Such a two-fold division of programming logic is a natural-fit to typical sensor networking operation which makes sensing and topological modalities accessible to the user.

Table 4.2: Commonly used Byte-Codes in Nano-CF

Instruction Type	OpCode
Declare/Assign:	`DECL, AEQ`
Arithmetic:	`ADD, SUB, MUL, DIV`
Comparison:	`GE, LE, EQ, GT, LT`
Constructs:	`IF, ELSE, FOR, GOTO`
I/O:	`SET, GET, TOGGLE, CLR`
Nano-RK:	`SEND, RECV, WAIT`
Macros:	`LABEL, SECTION, END`

4.6 The sMapReduce Programming Abstraction

In this section, we propose *sMapReduce*, a programming abstraction to divide the network-level user program into explicit *sMap* and *Reduce* functions. Most of the sensor networking applications can be visualised as accomplishing two important and largely disjoint functions, *namely*: *i*) Sense and compute, *ii*) Forward and Aggregate. Isolating these functions at the programming abstraction level helps a programmer not only to visualize the network operation with ease, but also to implement complex application logic. The *sMap* operation maps the application *behavior* to the *structure* of the network. Hence, we call our approach *sMapReduce*, which stands for *structure-Map & Reduce*. *Structure* means the network topology and the configuration of nodes, including the hardware and software capabilities. By application *behavior*, we mean the expected functionality of the structure of the network of sensor nodes. The *Reduce* function handles the responsibility of data aggregation in the network-tree topology.

In this regard, several abstraction concepts from the field of distributed computer systems can be adapted to sensor networks. Sensor networking applications are typically less data-intensive but data are highly correlated to physical location as nodes are deployed to sense the environment. In addition, gathering data efficiently from the nodes in a multihop network requires aggressive packet scheduling and aggregation to reduce the radio and computation resource-utilization. Our proposed programming pattern is inspired from the MapReduce framework [121], a popular data-processing approach in distributed systems. MapReduce framework requires a programmer to divide the processing job into *Map* and *Reduce* functions. *Map* takes a key-value pair as input and converts it to

```
1 smap(service_name,list_of_nodes, period){
2   for each node in list_of_nodes
3     temp_value = gets(TEMP);
4     smap_emit(temp_value,node_id);
5   end
```

(a) sMap Function

```
1 reduce(data,list_of_nodes){
2   for each node in INNER.list_of_nodes
3     sum += data.temp_value; //AGGREGATION
4   end
5   return sum;
6 }
```

(b) Reduce Function

Figure 4.8: A simple example for collecting sum of temperatures from a wireless sensor network.

another intermediate key-value pair; *Reduce* does the job of combining this intermediate output from *Map*. Researchers have adapted MapReduce for processing of data on large sensor networks [122] with the premise that nodes carry huge amount of data and parallel computations are required in some applications. We take this concept a step further by mapping behavior to sensor nodes based on their logical and physical topology. The reduce concept is employed to implement aggregation logic over the network tree.

4.6.1 The sMapReduce Programming Pattern

In this section, we describe our proposed *sMapReduce* programming pattern for developing applications on sensor networks. As has been emphasized earlier, the bifurcated operation of sensor networks into behavior mapping and data-aggregation motivates a corresponding split in network-level programs. The user programs each application using two key functions: *sMap* and *Reduce*. The main objective of *sMap* is to associate sensing and decision-making jobs to the sensor nodes and the *Reduce* function handles collection of data through the network-tree while allowing the user to implement complex aggregation logic. *sMapReduce* is a higher-level programming pattern that maintains its expressiveness though disjoint *sMap* and *Reduce* functions.

A simplified example of an *sMap* function is shown in Figure 4.8a. The *sMap* function takes three

input arguments: `service_name` is the identifier of the application to be executed on the sensor nodes, `list_of_nodes` is a handler for data structure (or a database) containing topology and tree information of the nodes, and `period` is the period in ms at which the application repeats itself. The information about the sensor nodes, their hardware and location is compiled and stored in a data-structure during deployment. Most sensor network deployments are done manually, hence the mapping of physical location to a node id can be obtained in this phase. Once such mapping is available, a programmer can refer to a node through its unique id, or through the more abstract concept of physical location, logical location in the tree or even filtered based on sensor capability. The sensor capability can be specified by the availability of certain type of sensor, computation power or available battery capacity. Even in the case of dynamic topologies, the underlying routing and communication infrastructure can share the responsibility of providing frequently updated logical node location and topology information to the programming abstraction.

The functionality of nodes is decided in an *sMap* function. The user can make use of predefined library functions and programming constructs to create programs for the network. Some of the commonly used programming features have been listed in Table 4.3. Table 4.4 provides a list of operators to select a subset of nodes from `list_of_nodes`. In the example shown in Figure 4.8, we present a simple *sMap* application for collecting temperature data from all nodes in the network. The code for this application consists of a `for` loop to iterate through the list of nodes, an instruction using `get()` to read the temperature reading and then an `smap_emit()` to send the temperature reading along with the node id towards the gateway.

The *Reduce* section of the program is used to specify the aggregation scheme. A separate dedicated section in the program to perform aggregation provides more freedom and flexibility to implement data collection algorithms. The user can assign aggregation responsibilities to different nodes in the network tree. It makes it easier to overlay complex aggregation algorithms over the tree through higher-level abstractions for node addressing. This two-fold advantage is made possible by separating the sensing operation from the data-aggregation in *sMap* and *Reduce* sections. Figure 4.8b shows an example of a reduce function for calculating the sum of temperature readings obtained in the *sMap* section in Figure 4.8a. In this example, the `INNER` operator is used to select non-leaf nodes and the sum of the input temperature data is calculated over all nodes. The sum of these temperature readings can be used to calculate a more useful parameter such as the average temperature at the

Table 4.3: List of the sMapReduce programming constructs

Construct	Details
`list_of_nodes`	Data structure containing the list of nodes and their properties
`smap_emit()`	Data to be returned by each node
`get()`	Function to read sensor values into integers, takes sensor name as argument
`set()`	Function to set a GPIO Pin
`clear()`	Clear a GPIO Pin
`toggle()`	Toggle a GPIO Pin

Table 4.4: List of operators for selecting participating nodes from among the `list_of_nodes`

Operators	Details
`LEAF.`	Nodes on the periphery of the network
`INNER.`	All nodes except the leaf nodes
`HOP(k).`	All nodes at k^{th} hop from the gateway
`HAS(t).`	All nodes that have a `t` type sensor
`BATT(c).`	All nodes having remaining battery capacity of atleast `c`
`CONN(n).`	Nodes having at least `n` neighbors

gateway node. It is trivial to compute commutative operations like sum, maximum, minimum and count. Moreover, as a user can access the nodes according to their physical location or logical location in the network tree, more complex aggregations schemes can be implemented as well.

4.6.2 A Target Tracking Example

Target tracking is a common application in sensor networks and requires considerable coordination between nodes. We provide an example implementation using the signal strength of beacons from a target node in order to demonstrate the advantage of using *sMapReduce*. The application logic is split into *sMap* and *Reduce* functions as shown in Figure 4.10. The *sMap* function reads the Received Signal Strength Indicator (RSSI) values from received packets as shown in line 3 in Figure 4.10a. The *Reduce* function in Figure 4.10b triangulates the location of the target when an intermediate node receives information packets from at least three children nodes.

In the *sMap* function, each node generates four values: RSSI, corresponding time stamp, location of target and its own ID, as shown in line 5 in Figure 4.10a. The *Reduce* function receives these values from *sMap*, and evaluates an aggregation at all intermediate nodes. As shown in the example topology in Figure 4.9, only node `6` is able to collect three values required for triangulation of the target node `T` tracked by nodes `1`, `2` and `3`. The *Reduce* function in the example implements the majority of the application logic because only an intermediate node can process the RSSI information to estimate the location of the target. The reduce function also ascertains the temporal correlation of RSSI values from different nodes by checking whether all the time stamps lie within a window of size `win` (line 6, Figure 4.10b). It is evident from this example that *sMapReduce* performs aggregation close to the leaf nodes, reducing the communication and computation overhead near the gateway node. The `triangulate()` function in line 8 calculates the location of a target node based on the RSSI values and the coordinates of infrastructure nodes.

Approaches like TinyDB do not capture sensing or topological modalities, as the aggregation is handled by an automated query planner. The design of application logic might be simpler in TinyDB in many cases but *sMapReduce* allows a programmer more control with an implicit understanding of physical and logical location of nodes. More complex schemes like *Regiment* do not isolate the functionality from aggregation explicitly, which can complicate the application logic with sensing job being undesirably coupled to various points in the program.

4.6.3 Mapping Applications for Mobile Nodes

The sociometric badge [123] is an example sensor network application that targets assisted-living scenarios. The infrastructure for such an application is expensive to maintain once the nodes have been distributed and deployed. Adding additional features is likely to be impossible, and the lack of resources on specific nodes restricts the services that they can offer. The presence of mobile nodes also adds additional complexity with respect to node re-programming and data aggregation. The proposed programming pattern, *sMapReduce*, provides a flexible and extensible mechanism to develop such systems, which could consist of both mobile and static sensor nodes. In order to support such systems, *sMapReduce* introduces two new aspects: (i) multi-level *sMap* and *Reduce* function support, and (ii) periodic map execution. This enables system designers to use *sMapReduce* on sensor network systems with mobile nodes. Figure 4.11a shows an example system, where a mobile node called Fire-

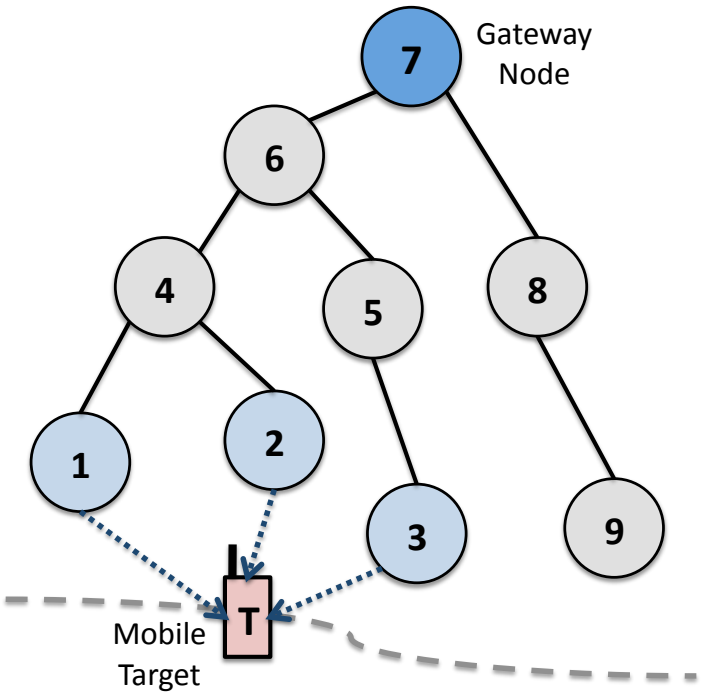


Figure 4.9: An example topology to demonstrate location tracking of a target node

Fly badge [124] is used to build the above-mentioned assisted living infrastructure. The FireFly badge could be hosting two location-based applications: (i) an emergency alarm application that needs to be loaded when the user is in a bathroom, and (ii) a schedule reminder application that needs to be loaded when the user is in a living room. The `smap_location` function is executed periodically, and it tracks the location of the FireFly badge so that `smap_location` can map the corresponding application to the badge. Then, `smap_service`, the second-level map function, will map `schedule_reminder` to the badge if the user is in the living room and `emergency_alarm` if the user is in the bathroom. Therefore, depending on the user location, a different application can be dynamically mapped on to the mobile node, and this enables programming of context-sensitive *sMap* or *Reduce* operations. This example thus illustrates a simple scenario where the multi-level mapping and periodic map execution features of *sMapReduce* can enable its use in networks with mobile nodes, where platforms such as [31, 32, 33] cannot be easily applied.

```
1 smap(target_track,list_of_nodes,period){
2   for each node in list_of_nodes
3     rssi_v = get(RSSI);
4     ts = get(time);
5     smap_emit(rssi_v,ts,node_id,loc);
6   end
```

(a) sMap Function

```
1 reduce(data,list_of_nodes){
2   for each node in INNER.list_of_nodes
3     if(data.loc != NULL)
4       return data.loc;
5     else
6       if(max(ts)−min(ts)<=win
7           && size(data.rssi_v) >= 3)
8         triangulate(rssi_v, loc);
9       else
10        return data;
11      end
12    end
13  end
14 }
```

(b) Reduce Function

Figure 4.10: A location tracking example using RSSI values of packets received by infrastructure nodes from a mobile target.

4.6.4 Features

The design of the *sMapReduce* programming pattern is based on the principle that a typical sensor network operation consists of two relatively disjoint functions. One associates a behavior to sensor nodes and another executes data aggregation over the distributed network. Hence, dividing the user program in explicit *sMap* and *Reduce* sections is a natural fit to sensor network operation. We provide below some features of the pattern to emphasize on the design decisions behind the *sMapReduce*.

Two-fold operation A typical sensor network operation consists of programming of the nodes and collection of data. These two are handled independently at different layers in the network. Further details of this operation are provided in Section 4.7.

Data correlation A sensor network is a distributed system where data of interest comes from the physical environment itself. Therefore, any computation on data should be conducted in the

```
1 target_service = smap_location( FireFly_Badge, list_of_nodes, period );
2 data = smap_service( target_service.name, target_service.nodes, target_service.
      period );
3 result = reduce(data,list_of_nodes);
```

(a) An example code for supporting mobile nodes

```
1 smap_location(service, list_of_nodes, period) {
2   for each node in INNER.list_of_nodes
3     if(node.location == bathroom)
4       smap_emit(emergency_alarm);
5     else if(node.location == livingroom)
6       smap_emit(schedule_reminder);
7     end
8   end
9 }
```

(b) Implementation of the first-level sMap function

Figure 4.11: An example of mobile node support

close neighborhood of the sensor node.

Programmer Support The explicit division of programs into *sMap* and *Reduce* sections allows the programmers to easily isolate the key functions, thus helping in easy inference and debugging of applications.

Balanced abstraction and control *sMapReduce* provides easy-to-use libraries and abstractions to deploy large-scale applications in addition to the ability to address individual nodes for fine-grained control to the user.

Expressiveness *sMapReduce* is a pattern derived from the operation of a sensor network, and it allows the programmer to conveniently map the behavior of sensing and aggregation to the network structure. The programmer can leverage subtle optimizations without much complexity in the application logic.

4.7 System Design

As previously stated, a typical operation of a sensor network involves two major components: *one* handles the programming of and coordination among nodes, and *another* governs aggregation of data

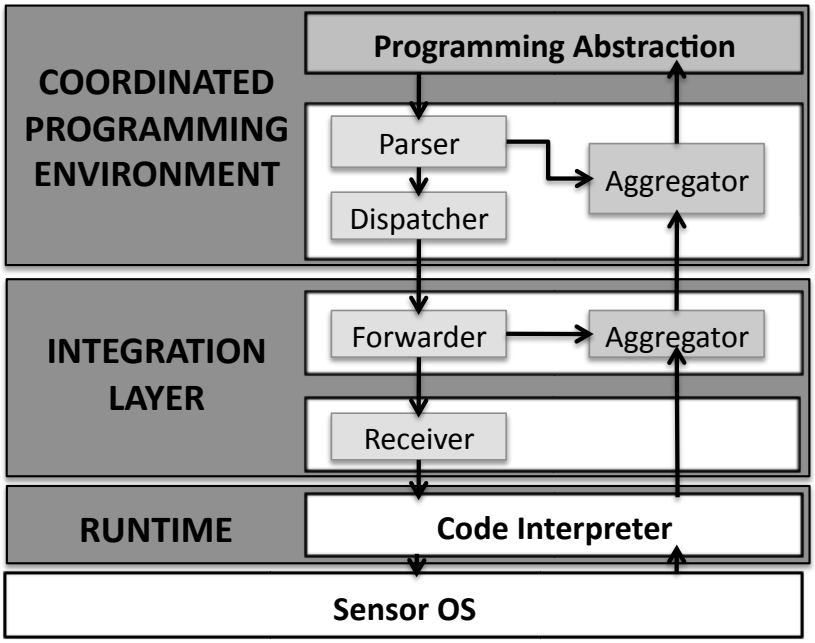


Figure 4.12: *sMapReduce* system architecture with three major layers to support a network-level programming abstraction

over the multi-hop network tree. We can conceptualize this two-fold operation as two independent planes that we call the *sMap* plane and the *Reduce* plane. Based on this concept, *sMapReduce* facilitates a programmer to distribute functionality in two separate sections.

4.7.1 sMap Plane

In the *sMap* operation of the system, the primary function is to assign specific behavior to each of the nodes in the network. *Behavior* in this case means all tasks executing on the node, along with communication handling and participating in data forwarding and aggregation. The layered structure along with *sMap* and *Reduce* operations is shown in Figure 4.13. The top-layer of this architecture is the programming abstraction for the user to create network-level programs in *sMap* and *Reduce* sections. The user-written program is compiled and converted into byte-codes to be executed on individual nodes. Byte-code execution implements the node *behavior* with the support from the sensor operating system. Byte-code is sent via the wireless network to each of the nodes, which is handled by a data-handler in the Integration Layer. The data-handler connects all sections of sensor networking

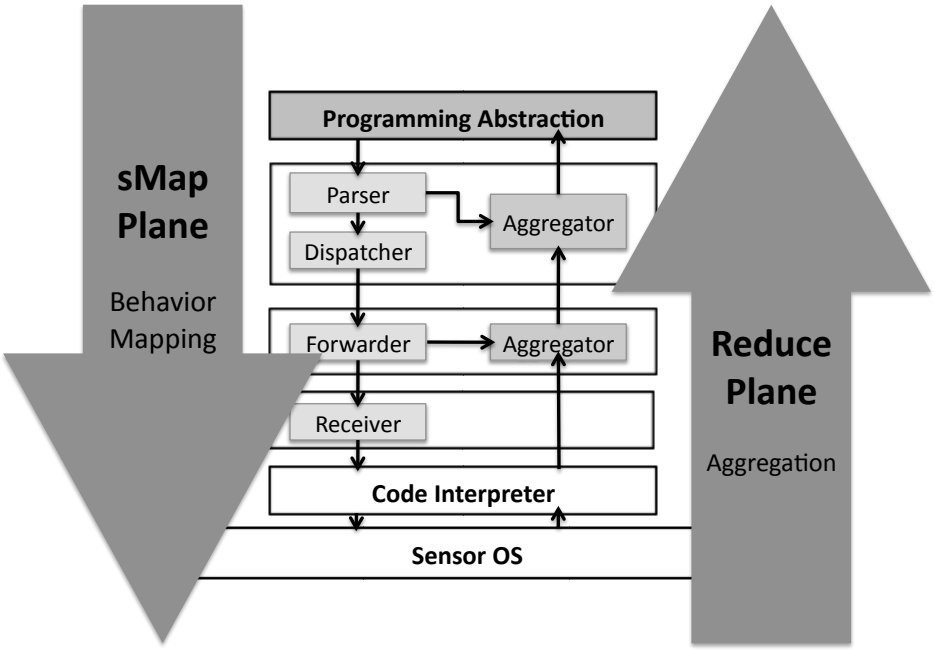


Figure 4.13: A figure showing the operation of *sMap* and *Reduce* planes. *sMap* operation involves top-down mapping of behavior to each node from gateway to leaf nodes, and *Reduce* handles data aggregation from leaf-nodes upwards

infrastructure spread over various layers, from the user-end PC at the top to the gateway node and intermediate nodes in the middle and to leaf nodes at the bottom. Once the byte-codes are delivered to each node according to their function, a code-interpreter converts them to sensor networking OS instructions. The byte-codes contain both the program to be executed and the aggregation scheme to be followed at each intermediate node. The main job of the *sMap* plane is to provide network abstraction and assign jobs to nodes while maintaining coordination among multiple applications and network hops.

4.7.2 Reduce Operation

Once the nodes receive the byte-code specifying their functionality/behavior, the nodes start the execution of the new application. The role of each node in the *Reduce* plane is also included in the byte-code where the intermediate nodes in the network tree help in aggregation of data. The aggregation of data is specified by the user in the *Reduce* section of the program. The role of aggregation can

be different for different nodes, depending on both the physical and logical location of the node in the multi-hop network. A leaf node should only forward locally sensed and computed information, and intermediate nodes may combine the data from their respective children nodes. In addition to aggregation, the *Reduce* plane should align, merge and schedule packets to reduce the overhead in communication. The *Reduce* plane is implemented through an aggregator module at every node and the Integration Layer supports the communication of data among different aggregator modules. Figure 4.13 shows how the *Reduce* plane overlaps over the right half of the system architecture. This split in the operation of sensor network justifies having explicit *sMap* and *Reduce* sections.

4.8 Evaluation of Nano-CF

One of the key goals of a macro-programming framework is to provide usability to the end-user. It should provide enough features to allow a programmer to program the network for most applications while being transparent to the underlying complicated details. This trade-off of complexity with transparency is not trivial to evaluate, and is highly dependent on the concerned end-user. Another major important aspect of the framework performance is the overhead in timing and energy consumption. In this section, we will provide the detailed evaluation of the Nano-CF framework with respect to energy savings, overhead of using a code-interpreter, and the usability of our programming language in terms of Source Lines of Code (SLoC).

Table 4.5: Power consumption information of the FireFly platform

Power State	Current	Voltage
All Active	24.8 mA	3.0V
Both CPU and Radio Idle	0.20 μA	3.0V
CPU Active	6 mA	3.0V
Radio Tx	17.4 mA	3.0V
Radio Rx	18.8 mA	3.0V

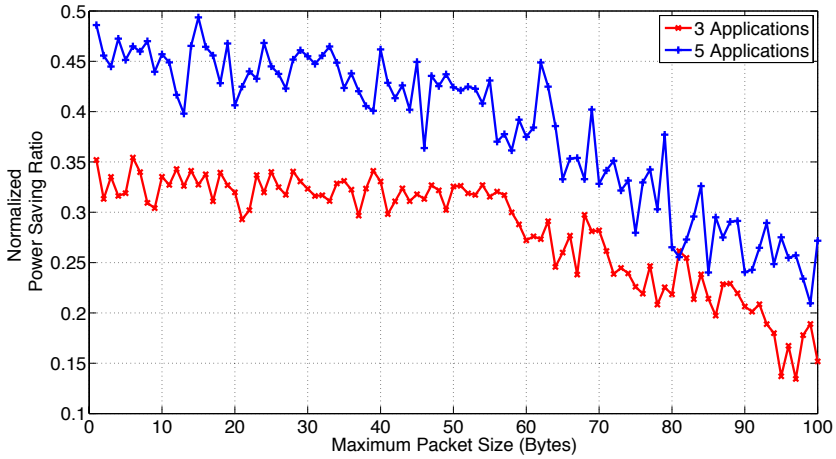


Figure 4.14: Applying RHS in packet delivery allows each sensor node save the energy by reducing the number of packets to be sent.

4.8.1 Energy Savings with Rate-Harmonized Scheduling

By clustering task executions and packet transmissions on the FireFly sensor nodes with RHS, we can reduce the energy consumed on each node. The power consumption of various components of a sensor node is provided in Table 4.5. As sensor nodes usually have low CPU utilization, we can guarantee even longer life expectation of each sensor node.

The framework supports delaying the packet transmission and hence combining the packets together, which yields significant power savings by using the transceiver for shorter durations at lower duty-cycles. This effect is shown in Figure 4.14. The figure is obtained by estimating relative power savings for randomly generated packets having a constant Period P_j with varying the maximum packet size from each application from 1 to 100. Every data-point shows the average after 50 iterations. When 3 applications are used, the energy consumption related to packet delivery can be saved up to 35%. In addition, if we use 5 applications, the amount of energy saving is increased up to 50%. As the maximum packet size increases, the effect of saving energy is decreasing. It happens because large packets may not be merged anymore. In addition, we can obtain opportunities to save more energy due to high possibilty of clustering packets from more number of applications. Aggregating packets together helps in reducing the number of packets transmitted in the network, which in turn reduces the channel contention and packet loss due to collision. More detailed evaluation of the network-layer performance is provided in Chapter 7, where a multi-hop networking protocol is

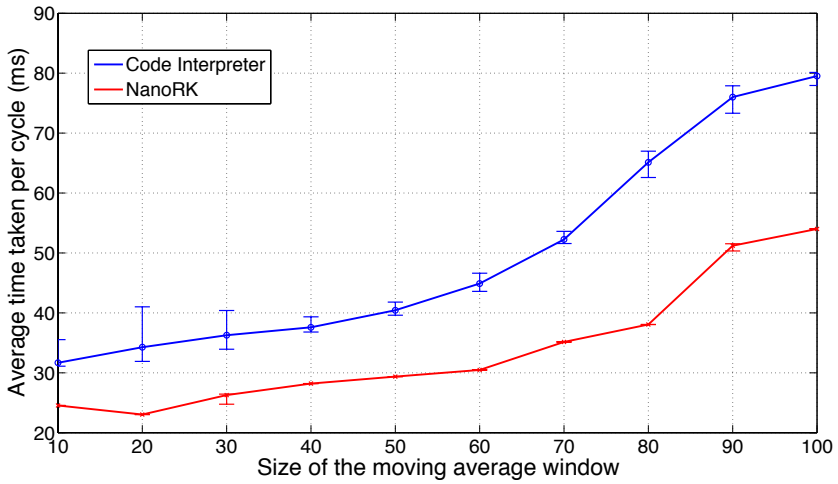


Figure 4.15: Code Interpreter performance comparison with equivalent code running on Nano-RK. The micro-benchmark used in this experiment calculates the moving average of light samples. This figure shows the average time taken per cycle by the micro-benchmark along with the corresponding error bars.

proposed that leverages the process of aggregating the packets, proposed in this chapter.

4.8.2 Performance Evaluation of the Runtime Environment

The runtime environment consists of a code-interpreter which executes the Nano-RK instructions corresponding to the received byte-code. We evaluated the overhead of the code interpreter task with respect to compiled code running directly with Nano-RK on the FireFly. The application we tested is a quite computationally-intensive task of finding a moving average of light sensor readings. The sensor samples over a window of given size are consecutively added and then divided by the size of the window. We observed the average time taken by the tasks to calculate the average with a varying window size. A larger window size means more cycles for processing in the task and hence a longer duration per cycle. Figure 4.15 shows the obtained results. It can be deduced from the plot that the code interpreter does not add much overhead to computationally-intensive tasks. We found the percentage overhead of the code-interpreter with respect to native Nano-RK code to be about 55.80%. Since, sensor networking tasks typically involve less computation, this overhead is quite acceptable. As shown in Table 4.5, the processor power is much lower than the communication power, where using Nano-CF results in 50% savings. In future work, we plan to investigate just-in-time compilation

techniques that can reduce the overhead of executing interpreted code.

Table 4.6: Comparison of number of lines of code from example in Figure 4.3

Application	Nano-CL	Nano-RK
Occupancy Monitoring	20	205
Temperature Collection	2	80

Table 4.7: Comparison of flash memory requirements for different applications running individually on a sensor node

Application	Nano-CL(Bytes)	Nano-RK(Bytes)
Occupancy Monitoring	35306	27932
Temperature Collection	35306	29324

Nano-CL allows the programmers to write their applications as composition of small services, where they do not need to consider the details of hardware setup and sensor configuration. We compare the typical number of lines of code a programmer is required to write for a particular application. Table 4.6 gives a comparison between the number of Source Lines of Code (SLoC) for the example in Figure 4.3 to similar applications implemented on Nano-RK. We can see the overhead in case of Nano-RK is more than a factor of 10. The number of SLoCs in Nano-RK programs are significantly higher because of the code required for task and hardware initialization. Comparison of memory footprints for these applications implemented using Nano-CF and directly on Nano-RK is provided in Table 4.7. The flash memory required by a Nano-CL program on a sensor node is the same regardless of the application, as the program is copied into RAM during in-network programming.

4.9 Summary

The ability to program a sensor network for multiple simultaneous applications using a macro-programming framework is a desirable feature. In this chapter, we presented Nano-CF, a framework which allows sensor network programmers to write applications on the sensing infrastructure with a

simple macro-programming language. We demonstrated the motivation behind supporting multiple independent applications through a macro-programming on a sensor network using the example of Sensor Andrew. With the proposed Nano-CF, we could save up to 50% of the communication energy when 5 applications are being used simultaneously on the sensor node. The code interpreter overhead was measured to be 55.80% on the average. However, the use of a code interpreter improves portability and maintainability. Furthermore, Nano-CF macroprogramming allows the user to create applications with significantly reduced complexity. Compared to developing an application directly on the sensor node operating system, we can implement the same function with only 10-15% of code lines.

In the next chapter, we exploit the further scope of intelligently combining multiple application's source code to remove any redundancy across tasks, based on the timing properties and user specifications. We plan to support the automated composition of multiple applications together into one or more by identifying common functionalities in the code written by users. This would allow our framework to be more efficient in the usage of resources at both the node and network level by combining the common subparts of tasks and data packets.

Chapter 5

Redundancy Elimination across Applications

A large percentage of applications for wireless sensor networks is designed around sensing the physical environment and transmitting a processed data value to the user. We call the paradigm for such applications as *Sense-Compute-Transmit (SCT)*. In such applications, there is a high possibility of redundancy across applications as they may contain several independent requests for sampling the same sensors. In this chapter, we propose an approach for eliminating this redundancy to save energy in the processor usage on each sensor node and the network.

Let us consider a simple case of a sensor network deployed across an office building with each node having a temperature and a humidity sensor. A building manager may be interested in collecting the temperature values from the sensors for a fine-grained temperature control, and a civil engineer may want to find the correlation between temperature and humidity for optimizing the building's HVAC system. Such applications can be executed concurrently on the sensor network infrastructure. Both the building manager and the civil engineering researcher sample the temperature sensor for their independent applications, which provides an opportunity for sharing the sensed value among both the applications. It turns out that reading a sensor value typically involves accessing the Analog-to-Digital Converter (ADC) module on the microprocessor, for converting the analog sensor value into a digital format, and storing into a register. This process of sampling a sensor can

consume about $2-3$ orders of magnitude more processor cycles than a simple memory-based instruction. With the increase in the number of applications deployed on a sensor network, the overhead because of sampling the sensors can also increase dramatically. Hence, by sharing the sensing requests among the applications, a significant percentage of resource usage and energy can be saved on a sensor node. We propose a solution able to achieve such energy savings through a compile-time approach. The challenges involved in such an approach are discussed next.

Computer science researchers have long focused on designing compiler optimizations to remove redundancies and dead-code in a program. Several simple optimizations are standard features in most modern compilers; complex features can also be enabled for specific optimizations based on overall program logic [125]. In general-purpose computing systems (e.g. desktop computers or datacenters), independent applications may have similar logic but it is very unlikely that they share the same data as well. This makes inter-application redundancy elimination a less-explored research area, as the possibility of energy savings is quite low. For instance, two independent users may want to use a distributed system to compute Fast Fourier Transform (FFT) over large datasets. Even though the computation module of FFT is the same for both the users, it is highly unlikely that the dataset will be the same as well. Hence, the provisions of sharing the same result among the two users may not be beneficial in terms of energy savings. In sensor networks, however, the data of interest typically is the sampled values of the physical quantities, and it is significantly more likely that different applications may require sampling of the same sensors. We show using our proposed approach that sharing those samples can achieve considerable energy savings.

As most sensing applications are periodic in nature and have low duty-cycles, eliminating redundant sections in case of mismatching periods can be difficult, and may not provide significant gains if elimination is carried out using simple temporal overlap detection. Secondly, the applications can sample the sensors multiple times at different intervals and in different order. Compiler support is a practical and effective technique for identifying such requests and optimizing them for finding better overlap. Finally, redundancy elimination at each node at run-time can add significant complexity to the scheduler on the sensor node. The scheduler in this case will have to pre-profile the execution of the program to identify the overlapping sections.

We propose a novel solution to the problem of finding overlapping sensing requests issued by network-wide applications created by independent users. We model each application as a linear se-

quence of executable instructions, and find a merged sequence of multiple applications through the use of well-known string-matching algorithms. In particular, we use the Longest Common Subsequence (LCS)[38] and the Shortest Common Super-sequence (SCS)[39] techniques. Our proposed solution creates a monolithic task-block resulting from the optimized merging of user applications with embedded scheduling information. This scheme is particularly advantageous in cases where the relative order of sensing requests is important, and simply caching the values may not help. One such case can be envisaged in an application where multiple sensors are sampled at different intervals but in a specific order to infer patterns of target behavior as may be the case in assisted living scenarios, or sensor-fusion based localization. We show that our approach can help in achieving about 60% average energy savings in processor usage as compared to the execution of several applications without eliminating such redundancies.

The organization of the rest of this chapter is as follows. First, we provide an overview of our approach in Section 5.1. Section 5.2 and Section 5.3 provide the details of the modeling of applications and the proposed redundancy elimination approach, respectively. We finally evaluate our approach in Section 5.4.

5.1 Overview of the Approach

We assume that the users develop network-level sensing applications using a higher-level programming framework. The application code written by the users can either be at an abstract network-level using a macro-programming language like Regiment [31] or it can use node-specific virtual-machines (for example Matè [32]). In both these cases, the programming framework creates node-level intermediate code based on the application logic specified by the user. Our approach is based on a machine-language like intermediate code, generally referred to as *bytecode*. The architecture of such a complete system is shown in Figure 5.1, where the user applications are converted into bytecode by a parser, such that each output instruction is either an indivisible subexpression or a special function for accessing the hardware (including sensing, GPIO access or packet transmission). Bytecode corresponding to all the applications are converted to a monolithic code by the *Redundancy Eliminator with Implicit Scheduler (REIS)* module. This monolithic code, which we call *REIS-bytecode* and ρ-code in short, is a merged sequence of all the applications with the redundancies eliminated according to

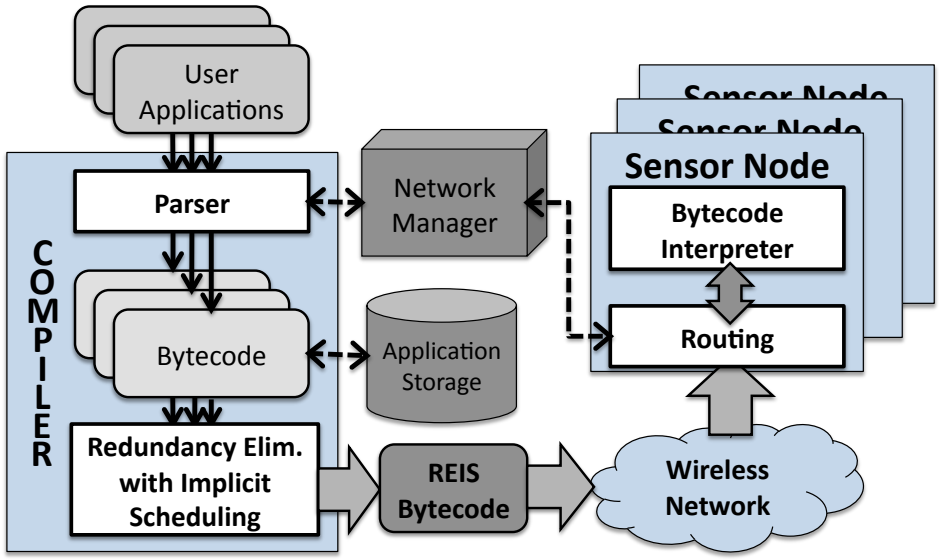


Figure 5.1: Overview of the approach for redundancy elimination

the temporal overlap of the sensing requests. REIS-bytecode is then sent over the wireless network to each sensor node where the applications are to be executed. A bytecode interpreter at the sensor node executes the received REIS-bytecode.

Our approach assumes that a data link-layer and a suitable routing layer are already implemented on the sensor node and our solution is transparent to it as long as end-to-end packet delivery is supported. A network manager module handles the responsibility of dynamically updating the routing tables, and maintaining the network topology information. As users issue applications to the system independently, our approach requires an application storage database to store the bytecode and merge them using the REIS module whenever a new application is submitted. The semantics of each user application is embedded within the REIS-bytecode such that the maximum sharing of sensing requests and radio transmissions is obtained. Bytecode from different applications share non-overlapping variable and address space, which removes any need for context switching, and the interleaving of bytecode provides an implicit schedule of execution.

The motivation behind the sharing of sensing requests can be justified based on the comparison of the time taken for reading a sensor sample into memory with a simple memory-based instruction. Figure 5.2 shows an oscilloscope capture of this comparison on a WSN platform with an Atmel AT-

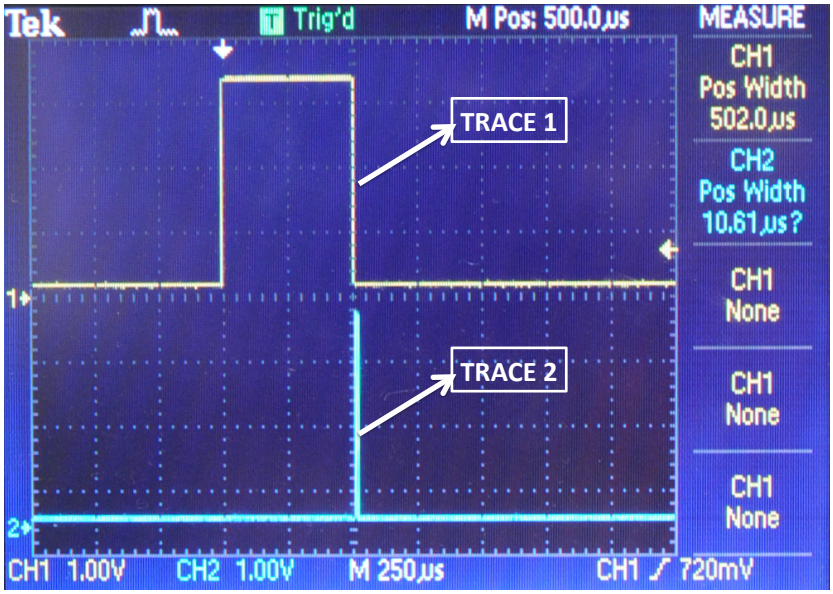


Figure 5.2: An oscilloscope screenshot showing the comparison of the time taken for reading a sensor sample against a memory-based operation

MEGA1281 processor. This comparison is obtained by toggling a GPIO pin just before and after the execution of a sensor sampling instruction (shown by Trace 1) and a memory-based loading of a 16-bit value into a register (Trace 2). The former takes about 500 microseconds but the latter instruction takes only 10 microseconds. Please note that this time comparison also includes the time taken for toggling the I/O pins. As the ATMEGA1281 ($8MHz$) processor on the sensor node has on-chip memory, a load instruction takes a maximum of 3 cycles that correspond to 375 nanoseconds. A majority of the time consumed in the case of Trace 2 is because of the pin toggling. Hence, a sensor sampling instruction consumes up to $\frac{(500-10)\times10^{-6}}{375\times10^{-9}} = 1306$ times more power. This factor, which we refer to as ϕ (*time-factor*), is specific to the platform and the operating system. However, the order of magnitude of ϕ can be assumed to be similar across most sensor network systems.

In addition, radios on newer System-on-Chip (SoC) solutions like the ATMEL ATmega 128RFA1 [40] support 2 Mbps data rate, compared to 250 Kbps from the commonly used CC2420 [53] radio for about 20% less power consumption. This implies that the power per packet can be reduced by a factor of 8, bringing the power consumption of the radio closer to that of the processor. Hence, optimizations at the processor level are bound to play a significant role in reducing total energy consumption,

in contrast to the majority of research efforts focusing mainly on energy savings at the radio.

5.2 Application Modeling

Our proposed optimizations are aimed at applications whose main goal is to sample sensors, process the sensor data for more meaningful results, and then transmit the results towards a gateway node through the network tree.

Each bytecode instruction contains a list of hex opcodes, and is of the form:
`<TYPE OP1 OP2 OP3 ...>`, where `TYPE` defines the kind of operation, and operand `OP<K>` can have specific usage based on the bytecode. For the sake of clarity, example formats of some relevant bytecodes are provided in Table 5.1. The specific implementation can vary based on the design of the Parser and the Bytecode Interpreter.

Table 5.1: Example bytecode structure for some relevant subexpression instructions

Operation	Opcode	Details
Sense	`S t VAR`	Sample sensor `t` and copy the value in VAR
Assign	`AEQ VAR1 VAR2`	Assign `VAR1 = VAR2`
Transmit	`T DEST VAL1 VAL2`	Transmit `VAL1` & `VAL2` to `DEST` node
Compute	`C VAR1 VAR2 VAR3`	`VAR1 := VAR2 'C' VAR3`

5.2.1 Conversion to a sequence of nodes

Most sensor networking applications are of the form: *Sense-Compute-Transmit (SCT)*, as the users are typically interested in sampling one or more sensors, processing the data from the sensors and collecting the processed results at a gateway node. Such applications can be modeled as a string of nodes where each node represents a sub-expression in the *bytecode*, β, as shown in Figure 5.3. S_t represents a sensing request (or a sampling request) for sensor type t, where t can be either be temperature (T), light (L), accelerometer (X, Y, Z) or any other sensor available on board. C denotes nodes with al-

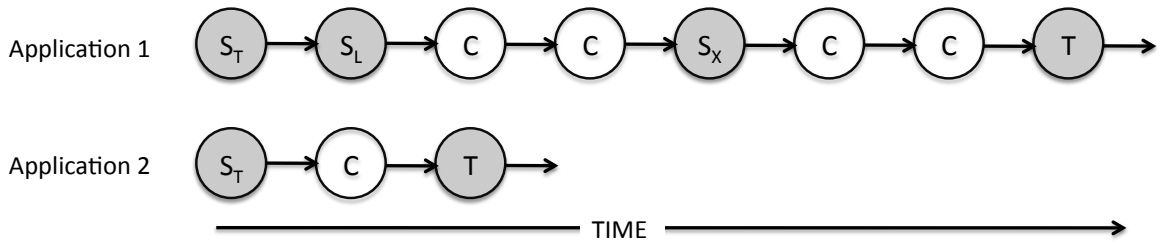


Figure 5.3: An example showing a linearized execution sequence for one instance of two applications. Application 1 samples three different sensors, and Application 2 samples the temperature and transmits its scaled-down value.

gebraic computation. As most sensor nodes typically have one kind of radio for communication, we use T to denote nodes corresponding to packet transfer via the radio. This conversion of bytecode sub-expressions to nodes is captured by the function `create_node()` in Algorithm 1. As algebraic computations are generally data-dependent, finding the overlap across C nodes is considerably less plausible. Moreover, there are no significant energy savings by eliminating such overlap, as these instructions typically consume a small (about 1 to 2) number of machine cycles, particularly on a sensor network platform having a RISC processor and on-chip memory. Hence S_t and T-type nodes participate in finding the overlap across applications, and are called *Anchor Nodes*.

Conditional statements in an application may not allow it to be converted into a linear string. We present the techniques for modeling applications having at least one anchor node inside the conditional statements in the next subsection. The conditional statements without an anchor node can be trivially mapped to a C type node.

5.2.2 Modeling Conditional Statements

As it cannot be known at compile-time which execution path can be taken in case of a conditional statement, it is not possible to create a ρ-code (REIS-bytecode) from the input bytecodes based on a linear application model as described in Section 5.2.1. We propose an algorithm to create a functionally equivalent code with a maximum possible number of sequential nodes, such that the conditional statements in the output bytecode sequence β_η are purely computational. Algorithm 1 provides a solution where the anchor nodes (sensing/sampling requests) are moved to before the beginning of

the outermost conditional statement in case of nested if-loops. An `assign` instruction is inserted in the place of the original instruction, which loads the value returned by the sensing request into the variable originally designed to read the output of sensing instruction, as shown in lines 19-21 in the algorithm. An example scenario is shown in Figure 5.4, where the original sampling request inside an `if`-condition is moved to before the outermost `if`-statement and the sampled value is stored in a temporary variable `var1_temp`. The original variable, `var1`, is assigned the value of `var1_temp` at its original location in the bytecode.

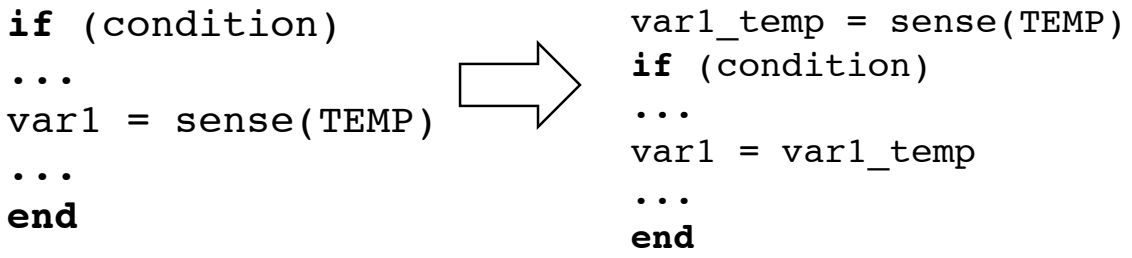


Figure 5.4: An example showing the modeling of an if-condition for redundancy elimination across applications.

Please note that the sensing requests are data-independent instructions; moving them to a previous point in the code does impact the application logic. Also, moving a sensing request earlier does not impact the sampled value, because there will only be a very small change in the exact sampling instant of the order of microseconds. Moreover, determining and enforcing an exact sampling instant is not always feasible in most sensor network operating systems. Hence, the sensed value is not compromised in terms of its temporal correctness.

5.2.3 Merging Packet Transmissions

It can be claimed that, for better power savings, the transmit nodes T should also be moved towards the end of the bytecode sequence to obtain better overlap of radio usage across applications. We, however, do not take such an approach in this chapter, since a solution is proposed in Chapter 7 to harmonize packet transmissions from different applications. Instead of transmitting whenever the applications request, the packets are queued in a local buffer and are transmitted at instants that provide maximum overlap of radio transmissions. As the radio is a shared resource among applica-

Algorithm 1: convert_app(A_i): convert an application to bytecode β_η

Input : A_i: A user created application
Output: (β_η, N_{an}): Bytecode node sequence, Number of moved anchor nodes

Parse A_i to bytecode β using the parser
INITIALIZATIONS:
$\beta_{\eta:} = \emptyset$; $if\_index := \emptyset$; $node := \emptyset$
$if\_depth := 0$; $N_{an} = 0$;
foreach *sub-expression* $\eta \in \beta$ **do**
    $i = IndexOf(\eta)$
    if η *is an if-clause* **then**
        $if\_depth++$;
        $if\_index \cdot append(i)$;
        $\beta_\eta \cdot append(\eta)$;
    else if $\eta = S$ **then**
        if *if_index·isEmpty()* **then**
            $index = i$;
        else
            $index = if\_index(i)$;
            $N_{an}++$;
        end
        $node := create\_node(\texttt{type}(\eta), var)$;
        $\beta_\eta \cdot insert(index, node)$;
        `// move` S `node before the beginning of outermost if-condition`
        $node :=\ create\_node(\texttt{assign}, var, \texttt{op2}(node))$;
        $\beta_\eta \cdot append(node)$;
    else if η *is an endif-clause* **then**
        $if\_depth--$;
        $if\_index \cdot pop\_back()$;
        $\beta_\eta \cdot append(\eta)$;
    else
        $\beta_\eta \cdot append(\eta)$;
        `// Non anchor nodes remain at the same relative location`
    end
end

tions, such a queue-based mechanism can help in achieving what is aimed by our proposed approach. Many other solutions (such as [126]) have also been proposed to optimize the network-wide schedul-

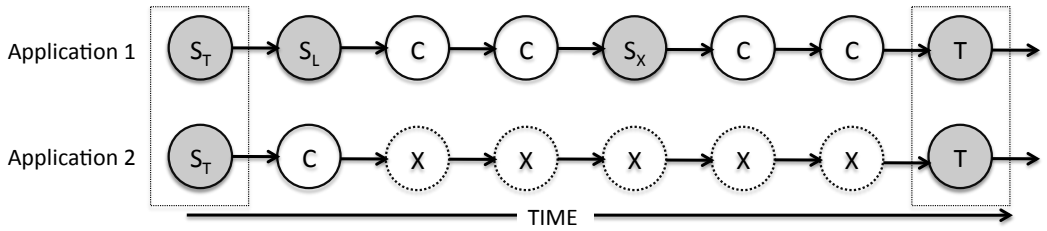


Figure 5.5: Application 2 modified to be aligned with Application 1 for sharing sensing requests and packet transmission (based on the example in Figure 5.3)

ing of packets.

5.3 Redundancy Elimination with Implicit Scheduling

5.3.1 String-Matching Algorithms

Once an application is modeled as a sequence of nodes as described in the previous section, the problem of finding overlapping sections among two or more applications can be reduced to that of finding a common subsequence between a pair of applications. The Longest Common Subsequence (LCS) is a technique commonly used to find the overlap between a pair of strings of symbols such that the relative order of common symbols is the same in both the input strings. LCS provides one such common sequence having the longest possible length. Consider the two following string sequences: `SENSOR` and `NETWORK`. The longest common subsequences are {`N,O,R`}, {`E,O,R`} but the Longest Common Sub-String (LCSS) would just be {`O,R`}. A longest common substring is always a subset of the longest common subsequence, but the opposite may not be true. There are some commonly available solutions [38] that are guaranteed to return *a* longest ordered subsequence between a set of input strings.

LCSS can help in finding redundant anchor nodes that appear consecutively in the input sequences. As an improvement over LCSS, LCS finds a subsequence with maximum overlap such that the relative order of nodes is not sacrificed. One or more of the input applications may be 'stretched' at various points, as illustrated in Figure 5.5 after applying LCS to the applications shown in Figure 5.3. An optimal merger of input sequences can be obtained by using an approach related to LCS

called *Shortest Common Super-sequence (SCS)*[39].

Definition 1. *Given input sequences X and Y, the shortest common super-sequence, $Z = SCS(X, Y)$, is the shortest sequence such that both X and Y are subsequences of Z.*

In the case of two input sequences, it is trivial to find the SCS if the LCS is known. For more than two sequences, finding the SCS is not a direct application of the LCS solution.

Algorithm 2: REIS(Γ): Generate a monolithic ρ-code with implicit scheduling from an input set of applications

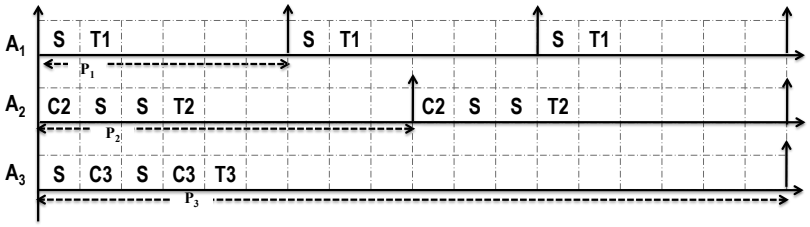
Input : Γ: a set of n applications $< A_1, A_2, \ldots A_n >$ each with period P_i for i^{th} application
Output: ρ-code: a monolithic bytecode sequence

```
// From Equation 5.1
```
$P_H := LCM(P_1, P_2 \ldots P_n)$
INITIALIZE:
for $i = 1 : n$ **do**
  $\beta_{new,i} := \emptyset$;
end
foreach *application* $A_i \in \Gamma$ **do**
  $(\beta_{\eta,i}, N_{an,i}) = convert\_app(A_i)$;
  for $j = 1 : \frac{P_H}{P_i}$ **do**
    `// create new strings`
    $\beta_{new,i} := concatenate(\beta_{new,i}, \beta_{\eta,i})$;
  end
end
ρ-code $= SCS(\beta_{new,1}, \beta_{new,2}, \ldots \beta_{new,n})$;

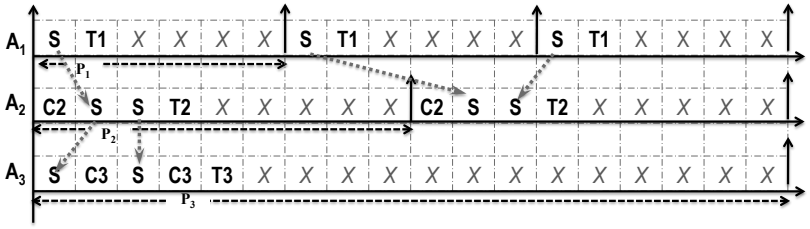
One important aspect of applications designed to operate on sensor networks is periodicity. Applications are typically designed as tasks that repeat periodically with low duty-cycles. Different applications deployed on a sensor network may have unequal periods. This adds further complexity to the redundancy detection and elimination across applications. Let us assume that an application A_i has a period P_i; the harmonizing period P_H is given by:

$$P_H = LCM(P_1, P_2, \ldots P_n) \tag{5.1}$$

where LCM stands for the Least Common Multiple of the input values.



(a) An example execution scenario showing three applications with different periods



(b) Process of locating overlapping sensing requests. The pattern repeats every hyper-period

ρ-code	C2	S	C3	S	T1	T2	C3	T3	X	C2	S	T1	S	T1	T2	X	X	X
δ		3		2							2		2					

(c) One possible output of the Algorithm 2, along with the degree of overlap of each shared sensing request

Figure 5.6: Identifying overlap in sensing instructions in three different applications and creating a merged ρ-code using Algorithm 2

5.3.2 Algorithm for generating a ρ-code (REIS-bytecode)

Let us consider a set Γ of n independent applications, where each application is denoted by A_i and $i = 1, 2, \ldots n$. The period of an application A_i is P_i. First, each application is converted into a sequence of bytecodes as described in Algorithm 1. The output of Algorithm 1 contains nodes within each periodic execution. As the periods can mismatch, the minimum length of time for which the overlap among two or more applications should be calculated is equal to the harmonizing period, P_H. A new sequence is created from each input bytecode sequence β_η by self-concatenating it $\frac{P_H}{P_i}$ times to create a new sequence β_{new}. After this operation, all the sequences are of an equal length of P_H. Thereafter, the Shortest Common Supersequence (SCS) solution is applied to find a merged sequence ρ-code from the concatenated input bytecode. This approach is expressed through Algorithm 2. This may result in the size of a merged application being quite large as the concatenated code corresponds

to P_H. However, it should be noted that there may be several repeating code blocks in the merged sequence that can be compressed significantly using simple compression approaches to save both the radio power and the memory footprint. This issue is beyond the scope of this work, and is not considered in our approach.

An example for demonstrating the merging of bytecode is shown in Figure 5.6. There are three input application bytecodes as shown in Figure 5.6a. Please note that all applications only sample one type of sensor for the sake of simplicity. The periods of the applications are different, and, in this example, $P_H = P_3$. Application A_1 consists of S and T nodes occurring consecutively with a period of 6 units. A_2 is a sequence $< C, S, S, T >$ with a period of 9 units, and A_3 is $< S, C, S, C, T >$ with a period of 18 units. Non-anchor nodes across different application sequences are considered as *dissimilar nodes*. For example, C in A_2 is not the same as C in A_3, hence they are represented as $C2$ and $C3$, respectively. The SCS algorithm considers only S-type nodes as common across applications and merges, such that the length of the merged sequence is the shortest possible. Figure 5.6b shows a possible alignment of the S nodes, and Figure 5.6c shows a merged sequence with the overlapping S nodes omitted. The degree of overlap δ for each merged node is also shown.

For n applications to be executed on a sensor node, each with Worst Case Execution Time (WCET) C_1, C_2, ... C_n, respectively, the total execution time of the input applications per hyper-period is :

$$C_T = \sum_{i=1}^{n} \left(\frac{P_H}{P_i} \times C_i \right) \tag{5.2}$$

where P_H is also the period of the ρ-code.

In the case of m overlapping instructions (anchor nodes), each with an execution time of E_i, the total execution time of the ρ-code is given by:

$$C_{T,\rho} = \sum_{i=1}^{n} \left(\frac{P_H}{P_i} \times C_i \right) - \sum_{i=1}^{m} ((\delta_i - 1) \times E_i) \tag{5.3}$$

where δ_i is the degree of overlap and is defined as the number of applications sharing a given anchor node.

5.3.3 Implicit Scheduling

The monolithic ρ-code obtained from the input applications is forwarded to the sensor nodes, where an interpreter executes it with a period equal to P_H. The design of the ρ-code is such that the constituent applications have explicitly non-overlapping variable space. The interpreter module has its own run-time stack to maintain its overall state, but it does not need to handle the responsibility of deciphering the individual applications inside the ρ-code. The schedule of each application is embedded in the sequence of instructions at the level of the hyper-period. If the total execution time without overlap, C_T, is less than the harmonizing period, the merged sequence ρ-code is guaranteed to finish the execution before the end of each period.

5.4 Evaluation

5.4.1 Comparison of Online vs. Proposed Solution

We compare the average power consumed by the radio of a sensor node with respect to the rate of re-programming of the network. The comparison is shown in Figure 5.7. It is intuitive that more frequent re-programming will consume more power. We compare the average power for the following scenarios.

1. The network is programmed using an online approach where a single application can be dynamically added.

2. Our proposed compile-time approach where a new monolithic ρ-code has to be sent to each node even if one application has been changed or added. The size of the monolithic ρ-code corresponds to 2 applications.

3. The ρ-code corresponds to 5 applications.

In this comparison, we assume that a node is only receiving application programming (bytecode) packets, and there is no other traffic in the network. We compare the average power consumption based on the assumption that the size of each application is equal to one data-packet of size 128 bytes and the power consumption of the radio is 56.4 mW (based on a CC2420 IEEE 802.15.4-compliant ra-

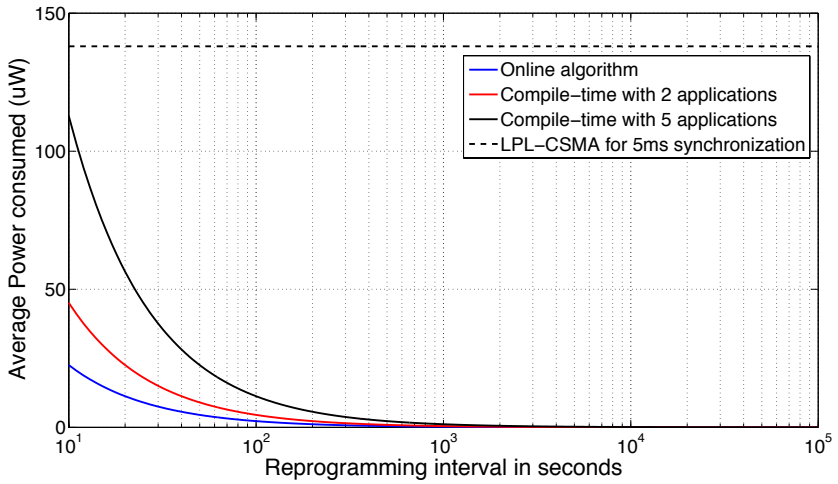


Figure 5.7: Comparison of average power consumed by the radio of a sensor node with respect to the rate of re-programming

dio). We notice that the difference of power consumed between the online approach and the compile-time approach diminishes fairly quickly. For instance, even if the network is re-programmed at a very high rate of every 100 seconds, the online approach will consume about $2\mu W$ on average, whereas our approach consumes about $11\mu W$ for a monolithic block of 5 applications. For more practical re-programming rates of the order of days or weeks, the absolute difference in average power consumption between our approach and an online approach will be negligible even for very power-constrained sensor nodes. To put this comparison of power consumption in perspective, the average power consumed by a basic LPL-CSMA (Low Power Listen - Carrier Sense Multiple Access) medium access protocol (MAC) is about $138\mu W$ for a background operation of maintaining time synchronization within 5ms accuracy [73]. We can therefore infer that even for a fairly frequent re-programming rate of every 100 secs, the power consumed is at least an order of magnitude lower than just the overhead of a light-weight MAC protocol. Even if the size of each application is bigger than one packet, the power consumed by both the online and the compile-time approaches for sufficiently low re-programming rates will be insignificant compared to the normal operation of the network.

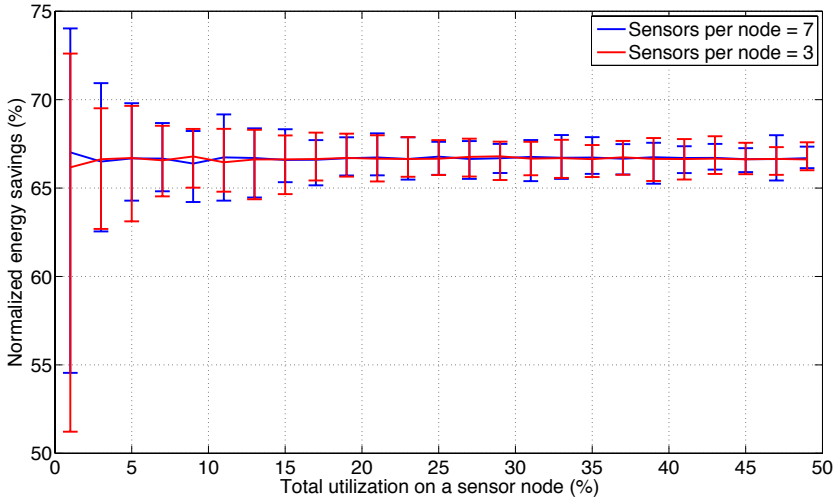


Figure 5.8: Energy savings with respect to increase in utilization of processor with different number of sensors.

5.4.2 Relative Energy Savings in Processor Usage

Energy savings in the processor usage after eliminating the redundancy in sensing requests can be estimated based on the degree of overlap δ by subtracting (5.3) from (5.2) and multiplying by the active power consumption of the processor. For the example scenario shown in Figure 5.6, the energy savings when the merged ρ-code is executed on the Firefly sensor platform [1] can be calculated as: $\Delta E = (2 + 1 + 1 + 1) * (490 * 10^{-3}) * (8.4 * 10^{-3})$. Hence, $\Delta E = 20.6\mu J$. On the other hand, the energy consumed by all applications running independently is approximately equal to $E_{orig} = 37.0\mu J$ if we ignore the negligible power consumed by other computation instructions. This corresponds to a significant 55% energy savings in processor usage for the particular example presented in Figure 5.6.

In addition to the above analysis, we conducted experiments to estimate the percentage power savings achievable from our approach in various cases. The results from these experiments are provided in Figures 5.8, 5.9 and 5.10. Each data point is collected by averaging across 50 iterations, and the error bars show the spread from the minimum to the maximum values over these iterations. These figures show percentage energy savings from our approach compared to the normal execution without any redundancy elimination. In this section, we consider only the processor usage because of the sensing requests, and we also assume that the energy consumption from other computations conducted on the processor is negligible in comparison. In Figure 5.8, energy savings are plotted in

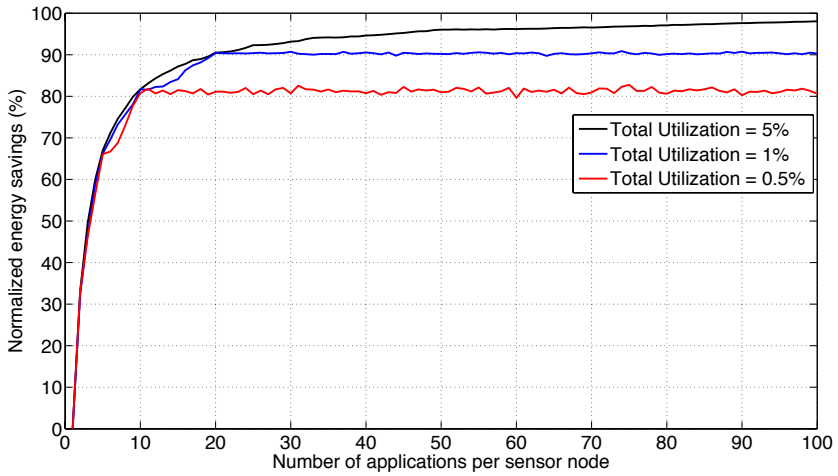


Figure 5.9: Savings in average energy with an increase in the number of applications.

the case of the execution of 5 randomly generated application strings on a sensor node and the total processor utilization is increased from 1% to 50%. Higher total utilization in our experiments arise from more sampling requests in the same ratio for each application. In this case, the average power savings remains more or less constant around 66%, but, with low utilization, the error spread is quite high. This is because, at low utilization, the number of sensing requests per application is low, and hence the possibility of finding redundancy is highly dependent on the type of the applications. As the utilization increases, the dependence of overall energy savings on application pattern reduces, as the chances of overlap are high anyway. When the number of applications deployed on a sensor node is increased, and the number of sensors per node is fixed to be 5, the energy savings increase as shown in Figure 5.9. This is because more applications can provide a higher degree of overlap, and hence more energy savings. The plot contains up to 100 applications just to illustrate the diminishing gains after a certain point. Such a large number of applications may, however, be impractical for most sensor nodes today. Figure 5.10 shows the reduction in energy consumption with respect to increasing the number of sensors on a node, and the average relative savings remains constant around 50% for 3 applications and 67% for 5 applications.

The intuition behind this behavior is the following. Even though the average degree of overlap, δ, may be lower for a larger number of sensors per node, equivalent energy savings are obtained. This is because there is a proportional increase in the types of sensing requests (anchor nodes) that leads to lesser overlap, since the total utilization is kept constant.

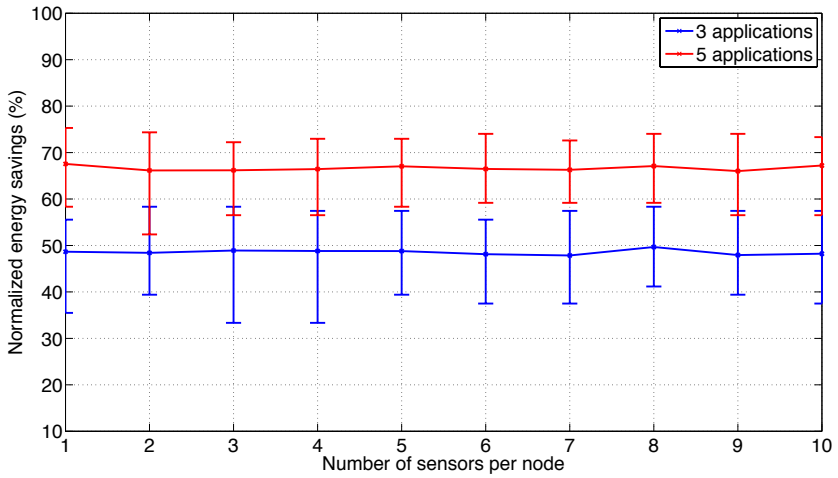


Figure 5.10: Percentage energy reduced with an increase in the number of sensors.

Overall, the achievable energy savings from the proposed approach is highly case-specific, but there is a high potential of energy savings if there are more applications or the utilization is high because of sensing-intensive workload.

5.5 Limitations

It can be argued that our application model is simplistic. It is, however, practical and it increasingly covers more and more scenarios of applications of large-scale sensor network deployments. Indeed, it does not support variable for-loops, and memory requirements can get prohibitive if loop unrolling is implemented. We leave an assessment of these issues to future work. Our approach is a compile-time technique, and therefore all applications are affected if one application changes or is added. On the other hand, a dynamic run-time approach can add significant overhead to the bytecode interpreter on the sensor node. In order for a run-time approach to efficiently eliminate redundancies across applications, pre-profiling of those may be required that can result in significant memory and processor overhead. Moreover, a compile-time approach is still beneficial if the rate of re-programming of the network is low.

5.6 Summary and Discussion

In this chapter, we have proposed and discussed a novel compiler-assisted scheduling approach that is able to identify and eliminate redundancies across applications in wireless sensor network infrastructures. Our approach models applications as linear sequences of executable instructions and we propose suitable algorithms for constructing such a model. We then show how it is possible to exploit and adapt well-known string-matching algorithms such as the Longest Common Subsequence (LCS) and the Shortest Common Super-sequence (SCS) techniques to produce an optimal merged sequence of the multiple applications with implicit scheduling.

As modern radio designs support higher data-rates for the same amount of power, the optimizations on processor power consumption become more relevant for energy-saving and increasing the lifetimes of sensor networks. On the other hand, with the increase in the number of applications deployed on a sensor network, the overhead because of sampling the sensors can increase dramatically. However, by sharing sensing requests among applications, a significant percentage of resource usage and energy can be saved on a sensor node. We demonstrate how our approach of using high-level optimization leads to significant network-wide resource savings, importantly energy. Our approach outperforms many other known techniques in the case of sensor node platforms supporting multiple sensors of multiple types. Our approach is highly predictable and its runtime is fairly simple: execution of bytecode with implicit scheduling. We show, based on experiments, that our proposed compile-time redundancy elimination approach can provide on an average about 60% energy savings on the processor with several simultaneous applications.

In the next chapter, we propose the design of a hierarchical system, where a monolithic REIS-Bytecode can be assigned to one of the tasks running on the operating system, rather than being the only executing block. Such a system can be modeled as an application of the classical bin-packing problem, where tasks can be clubbed together based on properties such as priority and memory requirements, providing scope for Quality-of-Service support in addition to resource optimizations.

Chapter 6

Hierarchical Assignment

In the previous chapter, we extended the advantages of a holistic over-the-air programming scheme by designing a novel compiler-assisted scheduling approach (called REIS) able to identify and eliminate redundancies *across* applications. In this chapter, we propose a hierarchical assignment scheme where the applications may be merged into multiple intermediate blocks, rather than one large monolithic block. Our evaluation shows that significant energy savings can be obtained by removing redundancies in sensor sampling, while meeting the resource constraints on the sensor nodes.

However, it might become impractical to combine all the applications into one large monolithic-block, as the size of the block might grow to be too large. As a further optimization, we present a hierarchical approach, where the tasks are merged into more than one intermediate task-block such that certain constraints regarding timeliness and memory usage are satisfied. The task-blocks are then executed as independent tasks on the sensor nodes. We show in this chapter that this hierarchical scheduling problem can be modeled in a way similar to that of a classical bin-packing problem. We provide certain approximations such that the problem can be reduced to that of quadratic programming and, hence, solvable with a reasonable time complexity.

If the relative sequence of the sensor sampling requests is not important, then a caching-based solution can also be a possibility, where the sensor readings are cached in memory along with a time-stamp. Whenever an application requests a new sample, the cached value is checked for its *freshness*, and if it newer than a threshold, the value is used as it is. Otherwise, a new sensor sample is taken and

```
1 typedef struct {
2   int16_t value;
3   time_t  curr_time;
4   } sensor_t;
5 sensor_t sensor;
6 //wrapper function definition
7 sensor_t get_sensor_val(SENSOR){
8   time_t ct;
9   ct = get_curr_time();
10  if ( ct-sensor.curr_time > THRESH){
11    // collect new sensor sample using
12    // the original function
13    sensor.value = get_sensor(SENSOR);
14    sensor.curr_time = get_curr_time();
15  }
16  else {
17    //return the cached value
18    return sensor;
19  }
20  return sensor;
21 };
```

Figure 6.1: Pseudo-code showing the wrapper function to collect sensor readings from a cache-based solution

provided to the application. Such a solution, devised using a wrapper function shown in Figure 6.1, is disadvantageous in a few major ways compared to our compile-time approach. Firstly, caching may not be practical in applications with fast-sampling rate. Secondly, it may not be directly applied in the cases where the relative order of samples from different sensors is important. As an example, an application may want to know when the light turned is on in a room by reading a light sensor, and then compare it with the reading from a motion sensor. Cached values in such a case may jeopardize the application semantics. If caching still needs to be used, the application behavior should be modified to be able to compare the time-stamps corresponding to different sensor samples requiring additional state maintenance. Thirdly, caching requires comparing of time-stamps which are typically 32-bit values; through simple experiments we found that it can take approximately $120\mu s$ to return the cached value (using the *else* path in Figure 6.1). This is significantly more computationally expensive as compared to reading a 16-bit value stored in memory with the Nano-RK [35] operating system running on a Firefly [1] sensor node that takes about $10\mu s$, as explained earlier and shown in Figure 5.2.

6.1 Hierarchical Assignment

In the approach described so far, all the user applications are merged into one monolithic task-block, which may have some disadvantages listed below.

- Different periods of the deployed applications may result in a large hyper-period, causing the size of the task-block to be prohibitively large. This not only increases the memory footprint, but also increases the number of packets required to transmit merged applications to the end-nodes.
- The applications may lose some timeliness because different parts of the applications may be executed with varying delays in the process of finding a better overlap.
- Some tasks may suffer significant delays in capturing the sensor sample and then using it, which may negatively impact application semantics.
- Data from some critical tasks (e.g., monitoring fire) may not be delivered with the required responsiveness.
- Including less sensing-intensive applications in the task-block may not conducive to energy savings.

We now present a hierarchical approach where we merge the applications into one or more intermediate task blocks instead of a large monolithic block. Let us consider a sensor node Operating System (OS) with support for multiple concurrent periodic tasks. The problem of executing the user-applications in such an OS can be represented as a hierarchical assignment one as shown in Figure 6.2. The goal is to strategically combine user-application tasks into multiple task-blocks such that several application requirements can be met. For example, merging applications with differing periods can cause the task-block to have a large memory footprint, and it may be beneficial to create more than one task-blocks such that tasks with similar periods are together. Similarly, creating task blocks from input tasks which share the same sensors may help in saving more energy.

6.1.1 Problem formulation

In order to find the assignment of tasks to intermediate blocks, we consider only the string model of the tasks. Other properties of the tasks, such as the semantics and timing behavior, are maintained

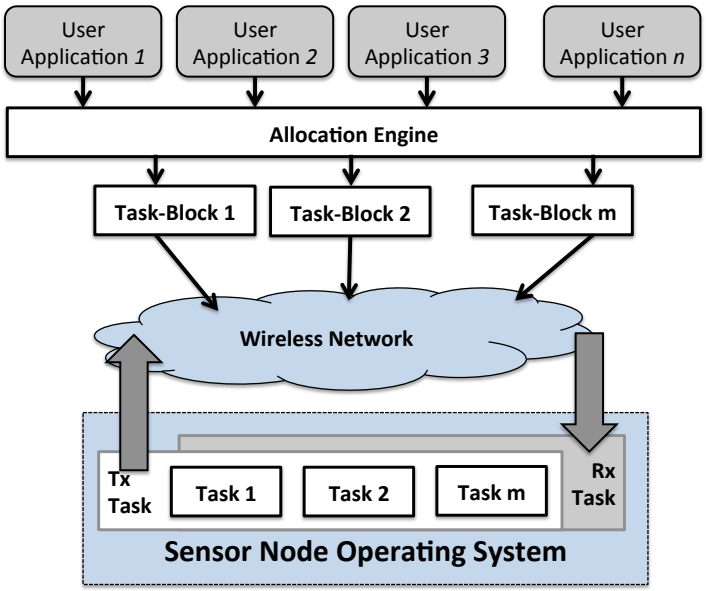


Figure 6.2: An architecture diagram showing the process of hierarchical assignment scheme for redundancy elimination.

as described earlier in Section 5.2. Let us assume that we have a set τ of n tasks deployed on the sensor nodes, and the corresponding bytecode string set is $\mathbb{B}$. The i^{th} string is represented by β_i, where $i = 1, 2, \ldots, n$. The tasks can be mapped to m task-blocks denoted by ρ_j, where $1 \leq j \leq m$ and $m \leq n$. The task-block is denoted by the symbol ρ because each task-block corresponds to a merged sequence of application strings, which we termed as REIS-bytecode or ρ-code as in Section 5.3. The challenge is to find an optimal mapping of n bytecode strings to m blocks such that the total energy consumption of all the applications is minimized within certain constraints. We model the problem as one of quadratic integer programming, which is an NP-hard problem with a solution space growing exponentially with respect to n. We also derive suitable approximations that provide a solution with reasonable time-complexity, but at the expense of exact optimality.

The total energy consumed by the system can be considered as the sum of the energy consumption of every application as follows:

$$E_{total} = \sum_{\forall i} E_{\beta_i} \tag{6.1}$$

Also, if the tasks are combined into task blocks, the total energy consumed is the sum of the energy

of all blocks. Therefore,

$$E'_{total} = \sum_{\forall j} E_{\rho_j} \tag{6.2}$$

The energy consumption of the j^{th} block can be obtained by subtracting the energy corresponding to the total degree of overlap across the tasks in the block from the total energy consumed by the tasks in the block, as follows.

$$E_{\rho j} = \sum_{\forall i \in \rho_j} E_{\beta_i} - \Delta_j \cdot e_s \tag{6.3}$$

where, Δ_j is the total degree of overlap in the j^{th} task-block, and e_s is the energy consumption of each sensing request. It results from (6.1), (6.2) and (6.3) that the energy savings after mapping tasks to task blocks can be estimated as:

$$E_s = E_{total} - E'_{total} = \sum_{\forall j} \Delta_j \cdot e_s \tag{6.4}$$

From (6.4) it is clear that, to maximize the energy savings, the total degree of overlap across all task-blocks should be maximum. The estimation of Δ_j is challenging as it involves finding the Shortest-Common Supersequence (SCS) for application strings with respect to each block. The degree of overlap for a given set $B \subseteq \mathbb{B}$ of bytecode strings is given as:

$$\Delta_j = \sum_{\forall \beta_i \in B} (\frac{P_H}{P_i}) L_i - L_\rho \tag{6.5}$$

where, P_H is the hyper-period of all the n tasks as defined in (5.1). P_i is the period of the i^th task. L_i denotes the length of β_i in number of bytecode instructions or nodes (Figure 5.3), and L_ρ is the length of the task-block obtained by finding the SCS of the input tasks strings. The challenge, however, lies in calculating the SCS for all combinations of β_i's so that the optimum combinations can be chosen for m blocks where $\sum_{j=1}^{m} \Delta_j$ can be maximized. There can be a prohibitively large number of such combinations that can make the optimization problem intractable. The optimization problem can consist of terms of arbitrary polynomial degree up to n. The problem will then require the partitioning of set $\mathbb{B}$ into m number of disjoint subsets, and thus, there can be an exponential number of such partitions that are typically calculated by the means of Bell number and Bell polynomials [127]. In order to be able to find an optimum assignment of tasks to task-blocks, the SCS's of all possible partitions may

need to be found along with solving the optimization problem. Hence, a major challenge lies in decoupling the problem from the calculation of SCS, so that it can be solved as a typical optimization problem with an objective function to be maximized while meeting some constraints.

Let us assume that the j^{th} block has h applications and $\lambda_{k,l}$ denotes the longest-common subsequence of the k^{th} and the l^{th} string in the set. The following theorem helps to approximate the optimization problem to one of Quadratic Integer Programming (QIP) with linear constraints. Such problems can still be NP-hard, but commercial solvers [128, 129] are available that and be used to solve them. Subsequently, we further approximate the model to a Quadratic Continuous Program that may be solved in polynomial time, but with loss of optimality.

Theorem 1. *The total degree of overlap over a set S of bytecode strings is less than or equal to the sum of the lengths of the Longest Common Subsequences (LCS's) of all the pairs in this set.*

$$\Delta_j \leq \frac{1}{2} \sum_{\forall k,l \in B} \lambda_{k,l} \quad s.t.\ k,l \leq n\ \&\ k \neq l \tag{6.6}$$

The equation is multiplied by a factor of half because $\lambda_{k,l} = \lambda_{l,k}$.

Proof. According to its definition, the Shortest Common Supersequence (SCS) of a set S of k strings contains all the elements of all the input strings but without any redundant elements. The SCS ρ of the strings in S is the shortest possible string such that all the strings in the set are subsequences of ρ. The shortest possible string can be found by removing redundant elements across all pairs of strings in the set S. The number of removed elements is equal to the sum of the lengths of the LCS's of all the pairs.

However, in this process, an element that participates in all possible pairs of LCS's of p ($2 < p < k$) strings, gets removed $\binom{p}{2}$ times rather than its actual redundancy degree of $p-1$. As $\binom{p}{2}$ is greater than $p-1$, the total degree of overlap is less than the sum of the lengths of the LCS's. The equality prevails when no identical elements occur in more than two LCS's.

□

As evident from the proof above, the exact value of the total degree of overlap is dependent on the elements of the constituent strings. Hence, it is not possible to isolate the problem formulation from finding the SCS, if an exact solution is to be found. The sum of the lengths of the LCS's of all the pairs

provides an upper-bound on the degree of overlap across all strings, and is used as an approximation to strategically combine tasks into task-blocks. Based on Theorem 1, we can now re-formulate the objective function so that the problem space can be reduced.

In order to formulate the optimization problem, we make use of *inclusion variables* denoted by X_i^j, where

$$X_i^j = \begin{cases} 1 & \text{if } \beta_i \text{ is assigned to block } j \\ 0 & \textit{otherwise} \end{cases} \tag{6.7}$$

Each task can only be assigned to one task-block to avoid repetition, which implies the following:

$$\text{for each } i, \sum_{\forall j} X_i^j = 1 \tag{6.8}$$

To find an optimal assignment of tasks to task-blocks, we quantify the degree of overlap by summing the pairwise degree of overlap multiplied by the inclusion variables, and the constraint in (6.8) accomplishes exclusion by making sure that each task is assigned to only one block. The goal of the optimization problem now is to find the values of the inclusion variables such that the total degree of overlap is maximized. Please note that the degree of overlap across two bytecode strings is the same as the LCS of the two, as described in Section 5.3. Hence, we can say:

$$\sum_{\forall \text{pairs} \in B} \lambda_{\text{each pair}} = \sum_{\forall a,b \in \{1,2,\ldots n\}, a \neq b} X_a^j X_b^j \delta_{a,b} \tag{6.9}$$

From Theorem 1, the objective function for the j^{th} task-block can now be written in an expanded form as:

$$\begin{aligned} \Delta_j \leq\ & X_1^j X_2^j \delta_{1,2} + X_1^j X_3^j \delta_{1,3} + \cdots + X_1^n X_n^j \delta_{1,n} + \\ & X_2^j X_3^j \delta_{2,3} + \cdots + X_2^j X_n^j \delta_{2,n} + \cdots + \\ & X_{n-1}^j X_n^j \delta_{(n-1),n} \end{aligned} \tag{6.10}$$

We can now create a general form so that the total degree of overlap D_{total} across all task-blocks can be maximized. Let $\mathbf{D}$ represent the $n \times n$ matrix where the $(k,l)^{th}$ element is $\delta_{k,l}$. The degree of

overlap of a string with itself is assumed to be zero, hence $\delta_{k,k} = 0, \forall k = \{1, 2, \ldots, n\}$.

$$\begin{aligned} D_{total} &= \sum_{j=1}^{m} \Delta_j \\ &\leq \frac{1}{2} \mathbf{X}^T_{n \times m} \mathbf{D}_{n \times n} \mathbf{X}_{n \times m} \end{aligned} \tag{6.11}$$

Most commercial optimization problem solvers, however, need the variables to be in a vector form. So, in order to vectorize $\mathbf{X}$, we need to replicate the matrix $\mathbf{D}$ into a matrix $\hat{\mathbf{D}}$ of size $M \times M$, where $M = m \times n$.

$$\hat{\mathbf{D}} = \begin{pmatrix} \mathbf{D}_{n \times n} & \mathbf{0}_{n \times n} & \cdots & \mathbf{0}_{n \times n} \\ \mathbf{0}_{n \times n} & \mathbf{D}_{n \times n} & \cdots & \mathbf{0}_{n \times n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{n \times n} & \mathbf{0}_{n \times n} & \cdots & \mathbf{D}_{n \times n} \end{pmatrix}_{M \times M} \tag{6.12}$$

Similarly, the vector form of $\mathbf{X}$ denoted by $\hat{\mathbf{X}}$ is:

$$\hat{\mathbf{X}} = \left[X^1_1, X^1_2, \ldots X^1_n, X^2_1, \ldots, X^2_n, X^m_1, \ldots X^m_n \right]^T \tag{6.13}$$

Now, the total degree of overlap from (6.11) can be rewritten as:

$$D_{total} \leq \frac{1}{2} \hat{\mathbf{X}}^T_{M \times 1} \hat{\mathbf{D}}_{M \times M} \hat{\mathbf{X}}_{M \times 1} \tag{6.14}$$

As the degree of overlap for any two strings is always positive, the total degree of overlap is maximized when all the tasks are assigned to one task-block. The problem, however, has other solutions if some other constraints come into play. As an example, if the maximum number of tasks in a task-block is fixed, then solving the above equation optimally assigns the tasks to specific task-blocks.

6.1.2 Constraints

The motivation behind the hierarchical assignment of tasks in this case is based on the fact that some constraints may not allow all the tasks to be merged into one monolithic block. These constraints may arise either because of some limitations on the sensor networking platform, or to satisfy timeliness

and criticality requirements of the applications.

Memory Constraint

As mentioned earlier, the length of a task-block corresponding to the hyper-period of all input applications may result in too large task-blocks for the amount of memory available on a typical sensor node. Also, the length of re-programming packets may become prohibitive. Therefore, the overall memory requirements of each block may be specified as a constraint as follows, where μ represents the maximum amount of memory that can be allocated to each task-block:

$$\text{for all } j, \sum_{i=1}^{n} \frac{P_H}{P_i} L_i X_{i,j} \leq \mu \tag{6.15}$$

where, P_H is the hyperperiod, and is equal to the Least Common Multiple of the periods of all the tasks. L_i denotes the length of the i^{th} task in number of bytecode instructions or nodes.

Number Constraint

The size of the task-block is also dependent on the number of tasks allocated to it. In order to simplify the constraints, an upper-bound, U, on number of tasks-per-block can be set. The constraint in this case can be written as:

$$\text{for all } j, \sum_{i=1}^{n} X_{i,j} \leq U \tag{6.16}$$

6.1.3 Objective Function

Based on (6.14), (6.7), (6.8), (6.15) and (6.16), we can formulate the optimization problem as follows:

$$\underset{\hat{\mathbf{X}}}{\text{maximize}} \quad \frac{1}{2}\hat{\mathbf{X}}^T\hat{\mathbf{D}}\hat{\mathbf{X}}$$

subject to:

1. $X_i^j = 0$ or 1
2. for each i, $\sum_{\forall j} X_i^j = 1$
3. for all j, $\sum_{i=1}^{n} \frac{P_H}{P_i} L_i X_i^j \leq \mu$
4. for all j, $\sum_{i=1}^{n} X_i^j \leq U$

The objective function is an upper-bound on the total degree of overlap as explained with Theorem 1, but can serve as a suitable approximation. This solution to the objective function provides an assignment where n tasks are allocated to m blocks such that the total degree of overlap across all the blocks can be maximized. Please note that Constraints 3 and 4 may or may not be simultaneously applied. Using any one of them makes sure that all the tasks are not merged into one task-block.

6.1.4 Continuous approximation

The objective function described in the previous section can be solved under the constraint that the inclusion variables are binary, and hence the problem is one of Quadratic Integer Programming. To reduce the time-complexity, the problem can be relaxed to a Quadratic Continous Program (QCP) where the inclusion variables can take continous values from 0 to 1 ($0 \leq X_i^j \leq 1$). The problem can now be solved in polynomial-time, but the solution fractionally assigns tasks to task-blocks. Simple heuristics can provide an integer solution, which may not be optimal, but is computationally inexpensive. We propose one as described below:

1. A solution is obtained by solving the QCP, where the tasks may be fractionally assigned to task-blocks.

2. The tasks in each block are sorted in a descending order of the values of inclusion variables.
3. Starting with the first block, the tasks are now assigned to blocks in a first-fit manner, until both the number constraint (Constraint 3) and the memory constraint (Constraint 4) are satisfied.
4. The same process is continued in the next block, until all the blocks are considered.
5. If all the tasks cannot be assigned to blocks, while meeting Contraints 3 and 4, the algorithm returns with a failure

6.2 Gains with Hierarchical Scheduling

The hierarchical assignment selectively merges tasks such that the degree of overlap is maximized within the given constraints of memory consumption or the maximum number of tasks allowed in each block. As shown in Section 6.1, Quadratic Integer Programming (QIP) can be used to compute an optimal assignment of tasks to task-blocks. We use the Gurobi optimizer [129] to solve the QIP. One example result with a maximum of 3 blocks is shown in Figure 6.3. We vary the number of tasks to be allocated from 4 to 13. Each block can have up to $(\lceil \frac{N_{tasks}}{N_{blocks}} \rceil + 1)$ number of tasks. For comparison, we also found an assignment using Quadratic Continuous Programming (QCP) where the inclusion variables $X_{i,j}$ can have real values rather than being integers. This reduces the computation time significantly, but the solution found may not be optimal. The values calculated for inclusion variables are rounded off, while making sure that the number of tasks per block does not exceed a $maximum$ threshold. The QIP computation time becomes impractical as the number of tasks increases. Even for 20 tasks, the computation time was in excess of 4 hours on a dual-core 2.7 GHz machine.

6.3 Related Work

In the domain of real-time scheduling on uniprocessor and multiprocessor systems, hierarchical approaches have been employed in the past for providing isolation, scalability and improving the task-allocation. A two-level scheduling approach for uniprocessor systems was proposed in [130] and there have been several similar studies (e.g., [131]) that allow the allocation of tasks on processors via intermediate *servers*, where servers represent abstract resources with pre-allocated budgets. In

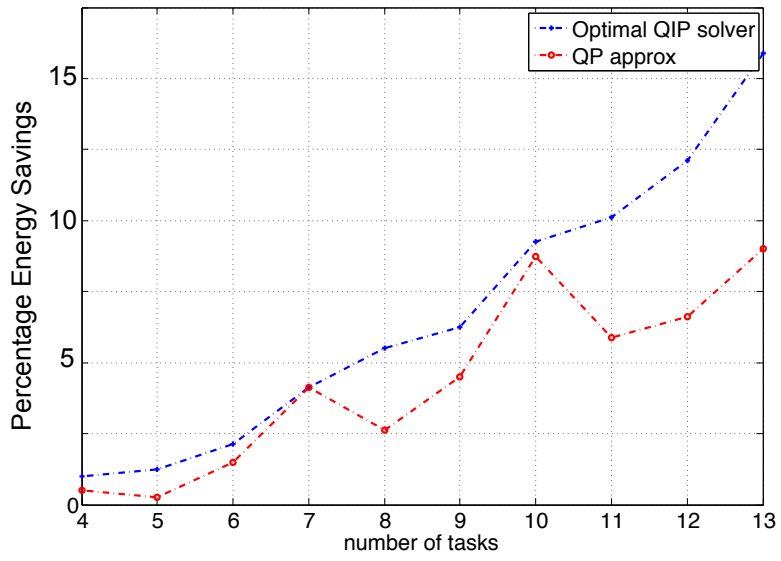


Figure 6.3: Relative energy savings in the case of hierarchical assignment. Better results are obtained using the optimal Quadratic Integer Programming (QIP) compared to an approximation obtained using Quadratic Continuous Programming (QCP or QP)

the case of multiprocessors, the task assignment is facilitated by virtually clustering the input tasks and allocating clusters as one entity [132]. Our proposed scheme is conceptually similar to such hierarchical systems, where tasks are merged into intermediate task-blocks to exploit energy savings while ensuring other important requirements like timeliness or memory footprint are fulfilled. The major difference between our approach and the above-mentioned schemes is in that, while hierarchical schemes are focused mainly towards providing hard timeliness guarantees, we focus primarily on reducing energy consumption by eliminating redundancy of sensing with soft real-time considerations.

6.4 Summary

In this chapter, we proposed a heirarchical approach that extends the inter-application redundancy elimination approach from the previous chapter. Instead of merging all the applications into one

monolithic block, the heirarchical approach consists of an optimization framework that creates a set of task-blocks such the resource requirements on a sensor node are met while maximizing the redundancy elimination. This heirarchical assignment scheme decouples the process of string matching from the overall optimization framework and reduces the problem to one of quadratic programming. We also proposed approximations that relax the problem from quadratic integer programming to quadratic linear programming at the expense of optimality but with a significant reduction in computation time.

Chapter 7

Network-Harmonized Scheduling

In the previous chapters, we proposed optimizations for reducing the resource consumption at the processor and sensor level by eliminating the redundancies across applications. However, there is still scope for optimizing the network behavior when multiple applications are executed on sensor nodes. It is often the case that applications release packets independently in the network, which can lead to excess energy consumption due to factors like increase in the number of packets, more frequent radio-switching and extra contention at the Medium Access Control (MAC) layer. The energy consumed in transmitting a packet from a source node to a destination node depends on many aspects, and with common MAC approaches, a packet may undergo contention at several points in a multi-hop network. This jeopardizes the deterministic behavior of the network and makes it very difficult to provide any timing guarantees. In this work, we present the *Network-Harmonized Scheduling (NHS)* approach, in which radio transmissions from multiple applications are coordinated across a multi-hop network by *harmonizing* them around periodic boundaries, while obviating the need for explicit MAC and routing layers. NHS is a simple and effective approach that is inspired by Rate-Harmonized Scheduling (RHS) [41] and applied to the context of multi-hop networking. By using NHS, it is possible to provide real-time performance guarantees in a multihop wireless network without requiring any central coordination.

Even if applications release one or more packets in a periodic manner, the overall packet transmission by a sensor node may no longer be periodic because of the mismatching periods of different

applications. In such a case, the total number of packets released by a sensor node grows proportionally with respect to the number of deployed applications. All the packets may suffer contention at different hops in the network if the underlying MAC layer is based on a carrier-sense mechanism. Therefore, multiple applications releasing packets independently may increase the overall resource consumption in the network. Moreover, the latency suffered by packets in a dense multi-hop network may become prohibitively large and non-deterministic. To overcome these issues, NHS aligns packet releases from different applications around periodic boundaries at each node, and leverages this periodic behavior to harmonize the transmissions at the network level.

NHS includes a light-weight protocol that groups periodic batched transmissions from different devices, such that the nodes can turn on their radios when other devices transmit. We first describe the protocol assuming that all nodes lie in a single broadcast domain. We further develop the protocol to support multi-hop scenarios, where we harmonize packet transmissions in a periodic manner without any global state maintenance. One of the major advantages of our protocol is that it includes an implicit link-layer mechanism, and, from its multi-hop operation, it can be inferred that dedicated route maintenance is also not required. Moreover, the protocol provides deterministic bounds on the end-to-end latency for packet delivery, and design parameters can be chosen such that the packet deadlines can be met for real-time applications. This characteristic emphasizes the usefulness of NHS in sensing and actuation networks with closed-loop operation, such as Cyber-Physical Systems [133].

We also analyze the performance of NHS with respect to parameters such as end-to-end latency, maximum utilization of the channel and average power consumption. The implementation of NHS is simple (~ 400 lines of code), and does not require the modification of application semantics. Applications only need to declare their period of operation and the maximum number of packets they may transmit in every round. Each node only maintains information about its neighbors, but can still achieve a performance (in radio duty-cycle) similar to that of Time-Division Multiple Access (TDMA). NHS requires only a few cycles to converge to a stable schedule, and does not need additional information exchange if the network remains static. Our approach can also optionally provide a contention slot for supporting mobile and intermittently connected devices.

The key features of our Network-Harmonized Scheduling approach can be summarized as follows:

- A harmonizing task is designed for sensor nodes that *batches* together the transmissions from

multiple applications by ensuring that packets are released into the network only at periodic boundaries.

- A protocol is proposed that coordinates packet transmissions in a network around periodic boundaries. For multi-hop networks, this protocol works in a distributed manner and pipelines the transmissions from successive hops to make sure that no collisions occur at any node.
- We implemented NHS[1] on the Contiki operating system, and we show through experiments that the protocol is suitable for real-time applications by providing deterministic bounds on parameters such as the end-to-end latency, channel utilization and radio duty-cycle.

The remainder of this chapter is organized as follows. We describe the state-of-the-art in the next section, and contrast NHS with other studies that also aim at optimizing the network operation. In Section 7.1, we outline the assumptions made in our approach and propose the model of the applications and the sensor network. The process of batching the packet transmissions using RHS (which inspired NHS) is explained in Section 7.2. The protocol to harmonize data from multiple nodes in a single broadcast domain is proposed in Section 7.3, followed by the description of the NHS approach for generic multi-hop networks in Section 7.4. In Section 7.5, we discuss various design parameters, Section 7.6 provides the details about our implementation of NHS on Contiki and Section 7.7 contains the results from experimental evaluation. In Section 7.8, we provide a brief discussion of potential applications of NHS, its limitations and different possible ways to address them. We finally summarize the NHS protocol in Section 7.9.

7.1 Model and Assumptions

In this work, we assume that several network-wide applications execute concurrently on a sensor network, and each application has a corresponding node-level task that releases periodic jobs on each sensor node. The set of all the tasks on a sensor node is represented by Γ. We assume that there are n tasks in Γ, and the i^{th} task is denoted by τ_i, where $i \in \{1, 2, ..., n\}$. The tasks execute on top of the Network-Harmonized Scheduling layer as shown in Figure 7.1. Every task releases an infinite number of jobs with a period of T_i. Without any loss of generality, we sort the tasks in an

[1]Our implementation of the Network-Harmonized Scheduling protocol can be obtained at [134]

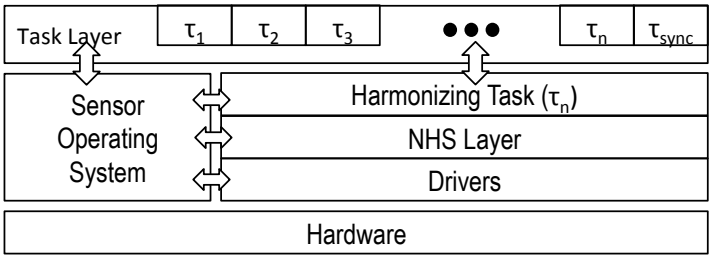


Figure 7.1: Layered architecture of Network-Harmonized Scheduling (NHS) protocol

non-descending order of their periods, and then assign serial indices to the tasks. Hence, it follows that $T_1 \leq T_2 \leq \ldots T_n$. Every job may sample the sensors, process the data from sensors or incoming packets, and release one or more packets towards a destination node. Assuming that the maximum number of packets a job of the i^{th} task can transmit is p_i, the time consumed by the packets for transmission is $w_i = p_i \cdot \delta$, where δ is the duration corresponding to one packet. As an example, the value of δ for IEEE 802.15.4 packets of size 128 bytes transmitted at 256 Kbps is $4ms$.

One of the tasks deployed on a sensor node can act as a simple clock synchronization service, τ_{sync}, that executes periodically. The management and exchange of MAC-level time-stamps and correction of clock-drift is the responsibility of this clock synchronization task which sits above the Network-Harmonized Scheduling protocol layer. As we will see in the next section, the harmonizing period (period of network operation) is chosen to be at least as small as the period of the most frequent task, allowing the synchronization task to comfortably operate at the desired frequency. It will be evident from the description of the protocol that each node only maintains a schedule according to its immediate neighbors and NHS only requires that a child node be synchronized with its parent.

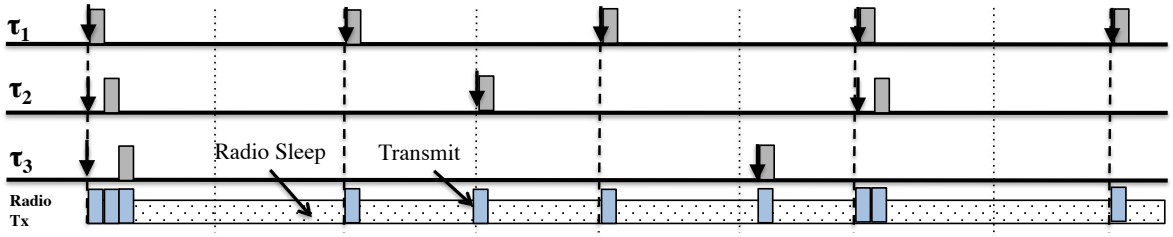
The nodes are assumed to have unique id's, and the number of nodes in the network is assumed to be bounded by N, and a multi-hop operation may be required to communicate from a root node to another node in the network, or vice-versa. Let h_{max} be the maximum number of hops required for connecting a root to all other nodes in the network, and Q denotes the maximum degree of connectivity in the network.

7.2 Rate-Harmonized Scheduling for Packets

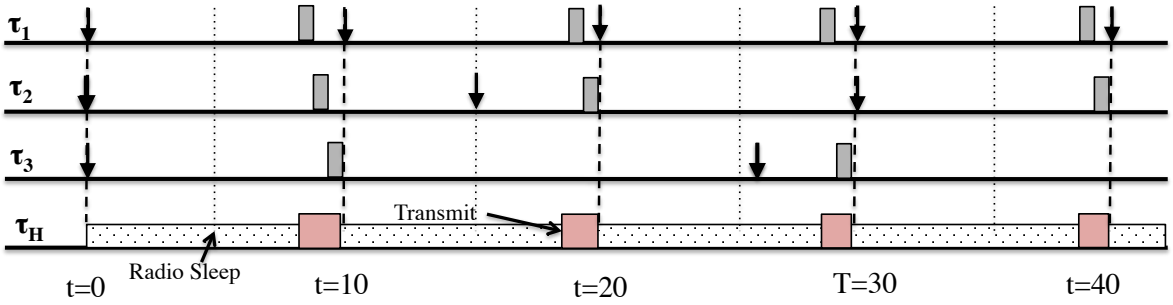
Rate-harmonized scheduling (RHS) [41] is a policy that optimizes the execution of tasks on a uni-processor system such that the job executions of all the tasks are aligned near the period boundaries of the task with the shortest period (the most frequent task). RHS saves power by removing inefficient switching in processor states, namely *active, idle,* and *sleep*, based on the observation that the power consumption in the sleep state is orders of magnitude lower than that in the idle state, but going to and coming out of the sleep state takes longer time. RHS makes sure that the task executions are harmonized and aligned in time, and the processor can optimally go to deep-sleep states more often and for longer time-spans. We adapt RHS in this work to align packet transmissions by different periodic applications on a sensor node, such that the overhead of radio switching can be avoided, and the packets are released into the network in a periodic manner.

With multiple tasks releasing packets at every T_i time units, the transmission pattern can be irregular as shown with an example in Figure 7.2a. The packets in the example are transmitted using the well-known Rate-Monotonic Scheduling (RMS) approach. On the other hand, the packets from various tasks are batched together with a harmonizing period $T_H = T_1$ as shown in Figure 7.2b. RHS is implemented using a simple queueing mechanism, where every job of all the tasks submits packets to a harmonizing task, τ_H, instead of directly copying them into the radio-buffer. τ_H then transmits all the packets in its queue with a period of operation equal to a *harmonizing period*, T_H. As the packets from a node are transmitted in a contiguous manner (back-to-back) as a *batch*, the number of radio switchings is reduced significantly.

An important distinction in the scheduling of packets is that preemptions are not possible once the packets transmission begins, whereas, most task-scheduling approaches on a processor, however, allow preemption. Harmonizing using the above mechanism converts intermittent packet transmissions from a sensor node into periodic batch-releases, and this periodicity is a fundamental building-block for further optimizations as described in the subsequent sections.



(a) Packet transmission from different tasks, if scheduled using Rate-Monotonic Scheduling. The task with a shorter period has higher priority. The number of independent packet transmissions over a time-window is the sum of number of packets from each task.



(b) The transmission schedule after harmonizing the transmissions. The harmonizing task (with a period of $T_H = T_1 = 10$ time-units) makes sure that the packets are dispatched in batches, never more often than the harmonizing period.

Figure 7.2: A task set with three tasks τ_1, τ_2, τ_3, with periods of 10, 15 and 26 time-units respectively, scheduled by Rate-Monotonic Scheduling and Rate-Harmonized Scheduling. Block arrows show the time-instants when the packets are released by different jobs of the tasks.

7.3 Single Broadcast Domain

Once the packet transmissions from multiple tasks are batched around the harmonizing period, we can design a distributed online protocol to align packets transmitted by multiple nodes in a single broadcast domain. Let us assume that the nodes have unique id's and, for simplicity, the root (or a cluster head) is already known among the nodes. The goal is to create a scheme where the transmissions from all the nodes can be gathered at the root in a periodically regular manner. The protocol works as follows. A simplified operation with a root node and 4 nodes with id's a, b, c and d is shown in Figure 7.3.

- Initially, all the nodes turn on their radios and listen to any incoming packets. The root transmits

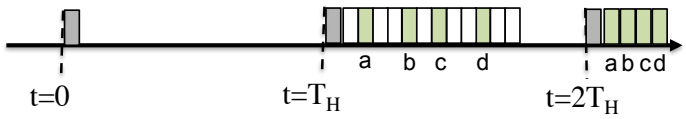


Figure 7.3: Aligning packet transmissions in a broadcast domain around periodic boundaries.

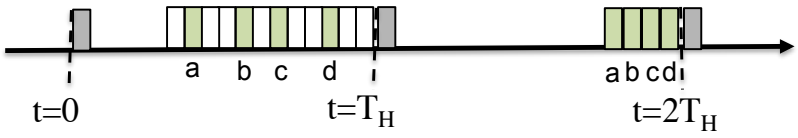


Figure 7.4: Aligning packet transmissions before the scheduled transmission by the root node, optimized for collection of data from all the nodes.

a beacon to initiate the protocol, with its node id and the harmonizing period.

- All the nodes listen to this beacon, and take note of the root id and the transmission period. The nodes locally create a schedule by assigning transmission slots as a monotonic function of node id, such that each transmission slot is unique to each node. This implies that the nodes with lower id transmit earlier, and those with higher id transmit later.

- Until the next period boundary, all the nodes listen to the medium after transmitting a chosen slot. As the node id's are assumed to be unique, it is guaranteed that there is no collision in any slot in the network.

- All the nodes listen to the medium during this period and they learn about the other nodes in the broadcast domain, and their respective slots. It is possible that there may be several empty slots in the schedule, and the nodes can then independently compress the schedule by removing the empty slots in the next cycle as shown in the second cycle of Figure 7.3 and Figure 7.4.

- From the next round onwards, the packets are transmitted in a compressed schedule such that the root only wakes up periodically for a duration equal to the total time required by all the slots. All the other nodes only wake up close to their slot in the schedule to transmit their data.

In this example, the protocol is designed for nodes to choose slots after the scheduled transmission by the root node in the next cycle. However, the protocol can also be modified such that the nodes

transmit *before* the next transmission by the root as shown in Figure 7.4. This is more beneficial in data-collection-oriented applications, because the root can receive the data and then react to the data in the same cycle. The difference in these two approaches is more pronounced in the multi-hop scenario (discussed in the next section), where the first approach is better suited for flooding and the second for many-to-one communications.

This protocol helps to remove the overhead of contention and random back-off as in the case of carrier-sense MAC protocols, and achieves TDMA-like timing efficiency without the need to maintain a global schedule. The width of each slot, denoted by σ, has to be large enough to accommodate batched packets, but short enough so that a Harmonizing Period can accommodate packets from all the nodes; that is:

$$\delta \sum_{i=1}^{n} p_i \leq \sigma \leq \frac{T_H}{N} \tag{7.1}$$

7.3.1 Addressing the Hidden-Terminal Problem

There can be cases of collisions in a multihop scenario due to hidden-terminal problems, because some nodes may occupy the same slot as a hidden terminal when the schedule is being compressed. This is avoided by ensuring that the root (or a parent) sends a message containing the list of successful receptions during the bootstrapping phase. Hence, every node becomes aware of other hidden terminals, and avoids choosing overlapping slots. If a node receives a packet from more than one parent node in the bootstrapping phase, then it chooses a parent which is closest to it, by considering the signal strength of the received packets. In this way, the network topology is generated by setting up parent-child links. If the topology changes, and a node does not receive a packet from its parent for a preset number of cycles, it choses a different parent within its neighborhood. By design, NHS is more suitable and reliable for static and nomadic sensor deployments than mobile and intermittent networks.

7.4 Harmonization in a Multi-Hop Network

We now extend the above approach to a multi-hop scenario where all the nodes have to send their data to a sink in the network. The protocol described in the previous section is easily applicable to

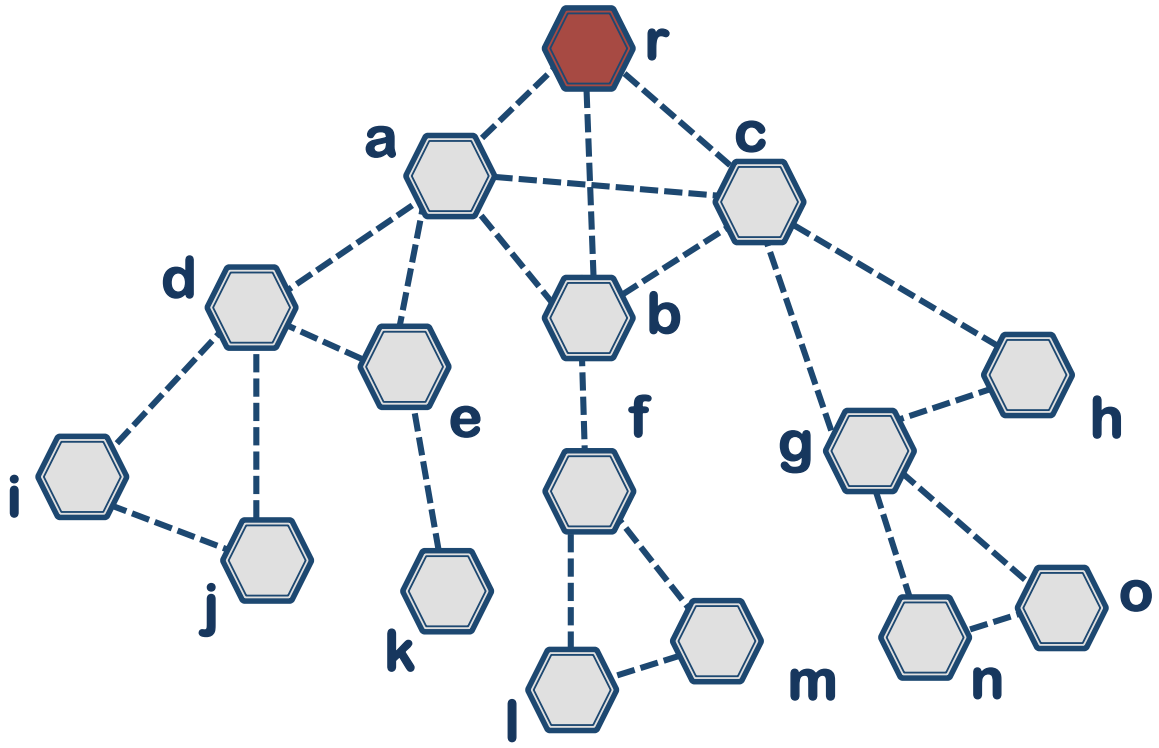


Figure 7.5: An example multi-hop network, with 15 nodes, and a root-node (r). Solid lines depict bi-directional wireless links.

multi-hop topologies with minor modifications, so that the possibility of collisions and packet-drops due to the hidden-terminal problem can be eliminated. The root initiates the protocol by broadcasting a *trigger* beacon, which is received by the neighbors of the root. Similar to the case of a single broadcast domain, the children nodes listen to the beacon, then choose slots as a function of their id's and then compress the schedule. The nodes at the second hop-level should not transmit until the schedule has been compressed, because the nodes at this level cannot listen to all the transmissions at the next level closer to the root. Each node in the network has information only about its 1-hop neighbors and its peers (siblings with the same parent).

The main goal of this protocol is to enable scheduled transmissions in a distributed manner, without requiring global knowledge of the network-topology and without explicit time-synchronization. The working principle behind the protocol is to ensure more than 2-hop distance in simultaneous transmissions in the network. Assigning slots to transmissions in a TDMA-based network is typi-

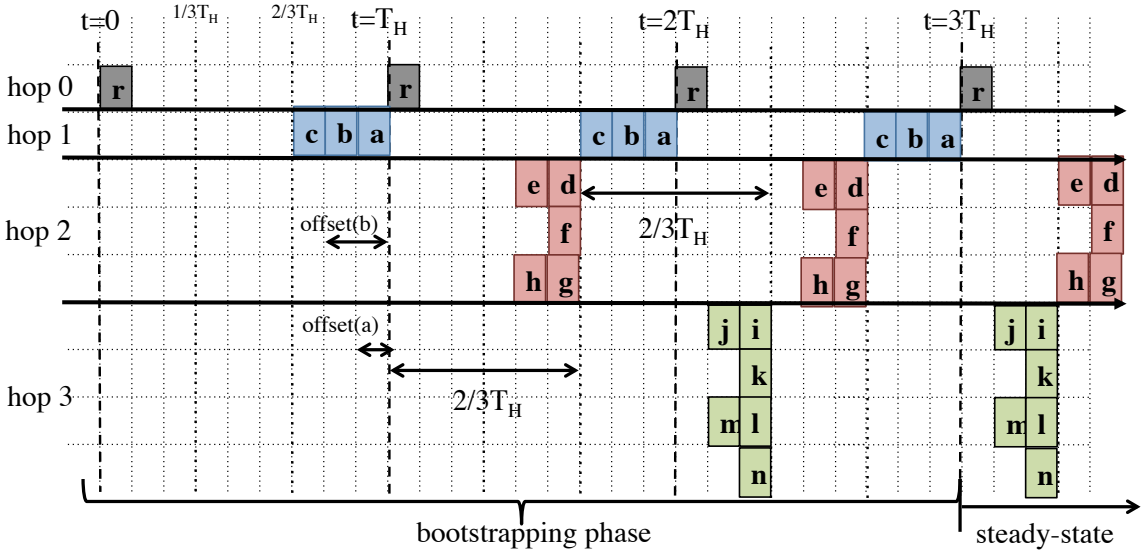


Figure 7.6: Timeline of transmissions from nodes at different hops, showing the working of the NHS protocol. The listening schedule is not explicitly shown, but the nodes can listen when the children and the parent nodes transmit.

cally accomplished by applying a distance-two vertex coloring graph. To maximize the throughput, the problem is equivalent to choosing the minimum number of colors [135]. In our approach, we achieve the required 2-hop distance by dividing each harmonizing period into three equal slices and nodes at consecutive hop-levels transmit only in non-overlapping slices. The number of slices can be chosen to be greater or equal to 3, and we call it the *cadence-factor*, ω.

The nodes at each hop choose slots to avoid collision using the approach described in Section 7.3 for the single broadcast domain case. In addition to the data to be transmitted, each node transmits a *NHS-tuple* as a part of the packet header. The NHS-tuple, denoted by λ, consists of η, the number of hops the transmitter is from the root, and ϕ, its offset in terms of the number of slots from the boundary of its section:

$$\lambda \equiv < \eta, \phi >$$

To illustrate the operation of this protocol, let us consider the example topology shown in Figure 7.5. We assume that data collection is the more important goal of the network, and the protocol is programmed such that the nodes transmit *before* the transmission of the parent node in the next

cycle, similar to the operation shown in Figure 7.4. The network consists of a root node r and several other nodes, namely $\{a, b, \ldots, o\}$. At the start, all the nodes turn on their radios and wait to receive packets. The root node r broadcasts a *trigger* beacon at $t = 0$, and nodes a, b and c receive this beacon. A simplified timeline of NHS operation corresponding to this example in shown in Figure 7.6, and the labeled slots imply transmission by a node. Nodes a, b and c transmit at the period boundary before the slot for the root. The transmissions are aligned around this boundary as explained in the previous section. For the sake of simplicity, the step of choosing non-overlapping slots is deliberately omitted in this example by assuming that the node id's are consecutive within a broadcast domain.

At the next period boundary, nodes lying within the first hop create a compressed schedule, and the transmissions from a, b and c are received by their respective children. Based on the λ values transmitted by them, their children nodes now also participate in the same protocol to find a local compressed schedule. The transmissions by the children nodes in the next cycle are carried out with respect to a future relative reference time T_s, which is estimated as follows:

$$T_s = \frac{2}{3}T_H + \phi$$

The offset values are shown for nodes a and b in Figure 7.6. The factor of $\frac{2}{3}$ is the key in dividing the harmonizing period into three sections, and making sure that the nodes in successive hops transmit *earlier* than their parents by one-third of the period.

Similarly, the children of the nodes in the second-hop, i.e. $\{i, j, ...o\}$ also transmit according to their respective T_s values and their transmissions do not overlap with any transmission in the first or the second hop. Using T_s, the nodes in the fourth hop would have chosen to transmit simultaneously with the transmission from the first hop, and so on. Simultaneous transmissions with NHS are guaranteed to have a hop-distance of three, making sure that no collision occurs at any receiving node. Designing a protocol with a *cadence-factor*, ω, of less than three can result in collisions at the receivers. On the other hand, if ω is chosen to be greater than 3, data from deeper hops can reach the root within one harmonizing period. The choice of ω provides a tradeoff between the latency suffered by a packet to reach from a leaf-node to the root and the maximum number of children a node can have. In general, the reference time T_s can be calculated with respect to ω as follows:

$$T_s = (\frac{\omega - 1}{\omega})T_H + \phi \tag{7.2}$$

In the steady state, the network operation is harmonized with respect to the cadence-factor (ω) and the harmonizing period (T_H). Transmissions by any node in the network are conducted only once every harmonizing period, but different hops are offset in a cyclic manner to avoid collisions. This approach improves the end-to-end latency in a fashion similar to that of pipelining. The nodes at the j^{th} hop now listen only in the windows where the nodes in the $(j+1)^{th}$ or $(j-1)^{th}$ hop are going to transmit. By listening to the nodes in the previous hop-level, the protocol ensures that the communication from the root node to leaf nodes is also possible. It can be observed that the data from ω number of hops can reach the root within one cycle, once the protocol reaches the steady state. The packets in the opposite direction can reach the root in a number of cycles equal to the number of hops. The network operation can be configured to be more responsive in either direction. If data collection is the main goal, then the described design is appropriate, otherwise the children nodes transmit just *after* the next transmission by their parent to enable fast delivery of data from the root to the leaf in case of flooding applications. This asymmetric operation is practical for most data collection applications, since the reverse data channel can be used for network maintenance, acknowledgements, and other similar functions.

The NHS protocol is distributed by design and maintains very little state. The primary benefit from this approach is that the transmissions are harmonized around periodic boundaries, and packets do not suffer from contention. The radios on the nodes only need to be turned on in a periodic manner for a small timespan, which considerably reduces the radio-switching overhead. The proposed protocol does not aim to achieve high throughput, since the goal is not to maximize the number of possible simultaneous transmissions in the network. That optimization problem requires global knowledge and has been solved in the past using graph-coloring approaches [82].

7.5 Real-Time Performance

So far, we have described the operation of the NHS protocol based on parameters such as the harmonizing period (T_H), cadence-factor (ω) and node degree (Q). The choice of these parameters can directly or indirectly impact the resource consumption and real-time characteristics of the network. The following sections introduce an analysis that allows us to reason about the trade-offs between the end-to-end latency and parameters: T_H, ω, and Q.

The observation that NHS parameters can be selected such that packets are always delivered before the next transmission, enables offline guarantees on end-to-end packet delivery deadlines, as presented in Section 7.5.2.

7.5.1 End-to-End Latency

The worst-case end-to-end latency that may be suffered by a packet from the time it is released by an application to the time it is delivered at the root node consists of the delays occurred due to the packet-batching and the latency in the multi-hop network. Based on the design of our approach for batching the packets from multiple applications, the worst-case delay a packet can suffer at a node is equal to the harmonizing period, T_H. The worst case happens when the packet is released by the application just after the end of a T_H cycle, and it can only be released into the network near the end of the next period boundary, as evident from Figure 7.2b.

At the j^{th} hop, the maximum number of slots that can be occupied for transmissions is equal to the maximum number of children a node can have. Also, the worst-case network latency suffered by a packet from a node at the j^{th} hop to the root is

$$L_j = \frac{(j-1)T_H}{\omega} + Q\sigma \leq \frac{jT_H}{\omega} \tag{7.3}$$

Each node turns on its radio to listen to its children and its parent and then to forward the data in the next slot. Therefore, the duty-cycle of the radio operation at each node, Δ, can be estimated as:

$$\Delta = \frac{(Q+2)\sigma}{T_H} \tag{7.4}$$

The total worst-case latency (including the delay at the node) suffered by a packet released by a node at the j^{th} hop is given by:

$$L_{total} \leq T_H + \lceil \frac{j}{\omega} \rceil T_H$$

We define the *delivery factor* φ, as

$$\varphi = (1 + \lceil \frac{h_{max}}{\omega} \rceil)$$

Hence, the worst-case latency for a packet from the h_{max} hop level is:

$$L_{max} = \varphi \cdot T_H \tag{7.5}$$

If the harmonizing period is chosen to be less than half of the period of the most frequent task (i.e., $T_H < T_1/2$), and if ω can be chosen to be greater than the maximum number of hops in the network, NHS guarantees that a packet from any node in the network can always be delivered to the root before the next packet is released by that node, and this determinism allows NHS to provide real-time performance guarantees to applications as we will see in the next subsection.

7.5.2 End-to-End Deadlines

The Network-Harmonized Scheduling protocol can provide deterministic end-to-end latency that can be leveraged to meet the packet delivery deadlines specified by the applications. We assume that the applications specify relative deadlines given by D_i, where $i = 1, 2, ...n$, considered from the point of release of the packet. For ensuring that all the packets meet their delivery deadlines, the maximum latency should be less than the minimum deadline, such that:

$$min(D_1, D_2, ...D_n) \geq L_{max} \tag{7.6}$$

which implies:

$$D_{min} \geq \varphi \cdot T_H \tag{7.7}$$

The above equation shows that a packet originating within ω number of hops can reach the root node within two cycles of the harmonizing period. Conversely, the harmonizing period can be selected such that the deadlines are always met.

The choice of the *cadence factor*, ω, is also important in determining the latency suffered by the packets over a multi-hop path, as given by (7.3). An increase in ω also results in narrowing the offset in the transmissions from successive hops; hence, it may not be possible to increase ω beyond the point that the transmissions from nodes at a hop may not fit inside a time-window of $t_\omega = T_H/\omega$. The

number of slots in a time-window of t_ω can be calculated as:

$$n_\omega = t_\omega/\sigma = \frac{T_H}{\sigma\omega}$$

Assuming that the maximum node degree is Q, the number of slots in each t_ω should at least be equal to Q to accommodate all transmissions from all the nodes:

$$n_\omega \geq Q \Rightarrow \frac{T_H}{\sigma\omega} \geq Q \tag{7.8}$$

By eliminating T_H from (7.7) and (7.8), we can deduce that:

$$\frac{D_{min}}{\varphi} \geq Q\sigma\omega \tag{7.9}$$

We can now find a suitable value for the maximum number of children a node can have such that the minimum end-to-end deadline is met.

$$Q \leq \frac{D_{min}}{\varphi\sigma\omega} \tag{7.10}$$

As the packets are batched with the same harmonizing period, we can say that, if the minimum deadline is met, the larger deadlines will also be met. Equation 7.10 provides an upper limit on the network size, such that the real-time requirements of applications are met.

The maximum number of packets or batches to be transmitted by the nodes in the first hop is the cumulative sum of packets from all the nodes at all hopes. Hence, the width of a slot should be:

$$\sigma \geq \beta \sum_{j=1}^{h_{max}} Q^j = \beta Q \frac{Q^{h_{max}} - 1}{Q - 1}$$

where, β denotes the maximum duration of transmission of a batch from a sensor node. Now, the relation between ω and Q can be written as:

$$Q \leq \frac{T_H}{\beta\omega \sum_{j=1}^{h_{max}} Q^j}$$

Table 7.1: Various fields in the NHS packet.

Field	Description
ID	ID of the transmitter
Slot	Offset (ϕ), transmitter's slot with respect to its parent
Parent	ID of transmitter's parent
Hopcnt	Hop count (η), Transmitter's Hop level
Cycle	Current cycle in number of harmonizing periods
N_child	Number of transmitter's children
Child_k	ID of the k^{th} child
NHS Data	Data from all the deployed applications, delineated by application ID and length.

or,

$$\omega \leq \frac{T_H}{\beta \sum_{j=1}^{h_{max}} Q^{j+1}} \tag{7.11}$$

To ensure that the end-to-end latency for packet delivery is less than the harmonizing period, the value of ω should be greater than the maximum number of hops.

$$h_{max} \leq \omega \leq \frac{T_H}{\beta \sum_{j=1}^{h_{max}} Q^{j+1}} \tag{7.12}$$

If $\omega > h_{max}$, then the latency will be greater than T_H. The duty-cycle of each node in the network, given by (7.4), does not depend on ω. This implies that ω can be chosen to be as large as possible to reduce the end-to-end delay without incurring any energy-consumption overhead.

7.6 Implementation

We implemented the Network-Harmonized Scheduling protocol on the Contiki operating system, such that it replaces the Radio Duty-Cycling (RDC) and the Medium Access Control (MAC) layers of the Contiki network stack. The core of the NHS implementation is a simple state machine as shown in Figure 7.8. The state machine is the same for all the nodes except the root node. The protocol starts with all the nodes in the network waiting to receive a packet with their radios on. Whenever a node

receives its first packet, it registers the id of the incoming packet as its parent node, and continues to listen for incoming packets for another harmonizing period so that it can listen to its peers. The packet header in the NHS implementation is shown in Figure 7.7, which contains several fields as described in Table 7.1.

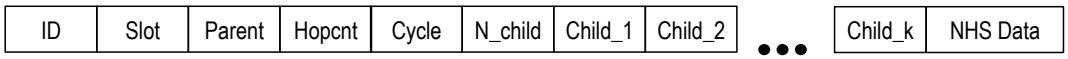


Figure 7.7: The packet header in NHS implementation

If a node receives a packet from a parent, it calculates the next relative time reference to transmit based on the received λ-tuple using the function `choose_slot_tx()`. This function calculates the reference time with the help of Equation 7.2, and conducts the slot selection algorithm among the neighbors of this node as previously described in Section 7.3. Then, the node goes to the sleep state, and wakes up at the time-reference Ts to transmit its data. Once the transmission is finished, the node goes to sleep immediately. The time instants for the nodes to wake up to listen are calculated using the function `ready_to_listen()`, that wakes up the nodes only when either the parent hop or the children hop transmit. The implementation of the Network-Harmonized Scheduling protocol for Contiki can be obtained at [134].

7.7 Experimental Evaluation

We implemented the Network-Harmonized Scheduling protocol for the Contiki [57] operating system for TMote Sky sensor nodes. In order to evaluate the performance of NHS with respect to energy efficiency, we compared the average radio duty-cycle achieved with NHS against an ideal TDMA approach. For estimating the duty-cycle of each node with the ideal TDMA, we assume that each node only turns on its radio at the exact instants to listen to its parent, its children and to transmit in its allocated slot. All other overheads of radio-switching and clock-synchronization are assumed to be negligible in the ideal TDMA estimation. The duty-cycle for each node in the case of ideal TDMA for a given topology is calculated offline by considering the network graph.

We measured the average radio duty-cycle per node for two different topologies, with varying values for the harmonizing period. Topology 1 is a linear topology as shown in Figure 7.9, where

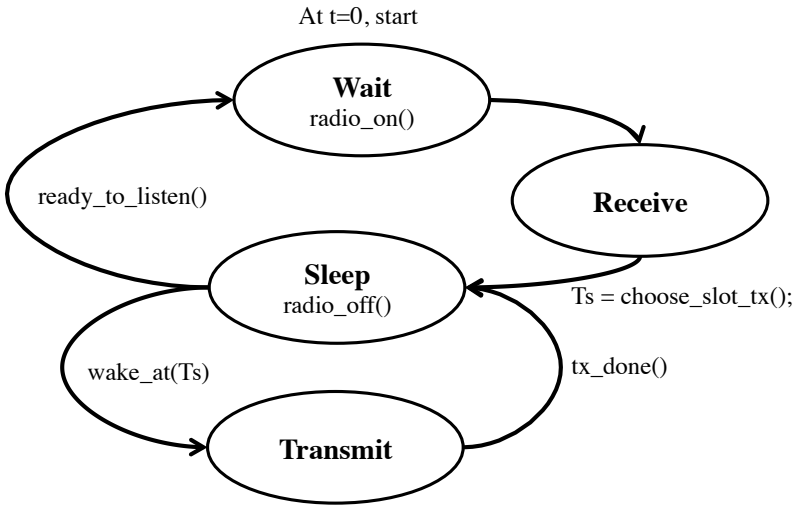


Figure 7.8: State machine showing the core of implementation of the NHS protocol at each node.

8 nodes are arranged in a linear topology with the maximum number of hops equal to 6. Topology 2 is a multi-hop tree topology shown in Figure 7.10. The values for duty-cycle are obtained after running the network for a duration of 600 harmonizing periods. The experiments are conducted for a large number of cycles to amortize the radio on-time in the bootstrapping phase, as the radio remains on continuously for the first few cycles. The results of the experiment are shown in Figure 7.11 and Figure 7.12. It can be observed from the results that the average duty-cycle in NHS is within 15% as compared to the ideal TDMA case. This overhead appears partly because the radio remains on for the first few cycles to identify the parents, peers and children, and partly because of the time consumed in the switching of the radio. It can be observed that the relative overhead compared to the ideal is larger for small values of harmonizing periods, which is about 30% for Topology 1 and about 12% for Topology 2, with a period of 1 sec. The overhead is larger for the linear topology because each node has only one parent and one child and the radio switching overhead accrues for each reception. However, for the tree topology, a node receives several packets from its children back-to-back, thus reducing the switching overhead. Overall, our evaluation shows that the average radio duty-cycle for each node in NHS is close to that of an ideal TDMA scenario, where NHS does not require a global knowldege of the network topology.

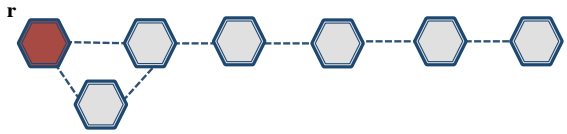


Figure 7.9: An example linear topology with 8 nodes.

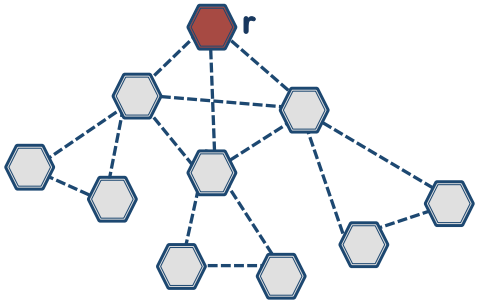


Figure 7.10: An example tree-like topology with 10 nodes.

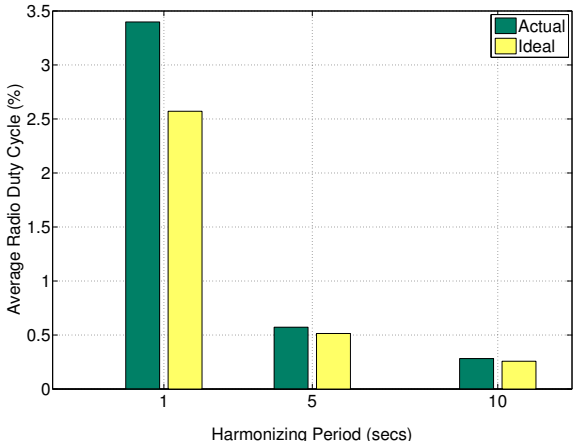


Figure 7.11: Average radio duty-cycle for the linear topology in Figure 7.9.

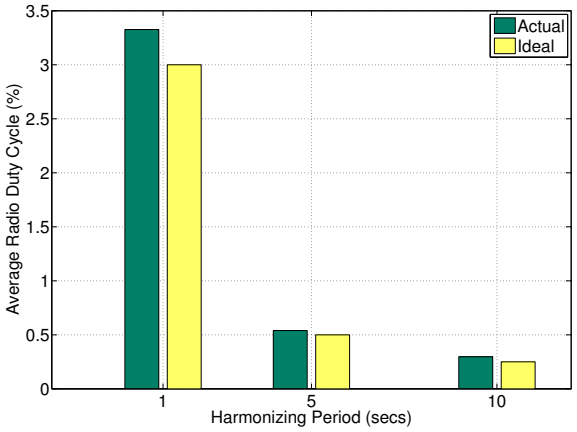


Figure 7.12: Average radio duty-cycle for 10 nodes in a multi-hop graph shown in Figure 7.10.

7.7.1 Real-Time Performance Evaluation

We also simulated the operation of the Network-Harmonized Scheduling protocol to evaluate its performance and validate the analytical results obtained in the previous section. The network topology for simulation consists of a set of nodes spread randomly with a uniform distribution in a 2-dimensional field of size $100m \times 100m$. A given number of nodes, N, is spread uniformly over the field. The number of nodes in each broadcast domain is automatically selected during the procession of the protocol. Corresponding to the number of nodes in the broadcast domain at a given hop level, successive hops are generated based on the nodes that lie within the radio transmission range. The simulation utilizes a time-driven execution, where each node is autonomously assigned a slot to

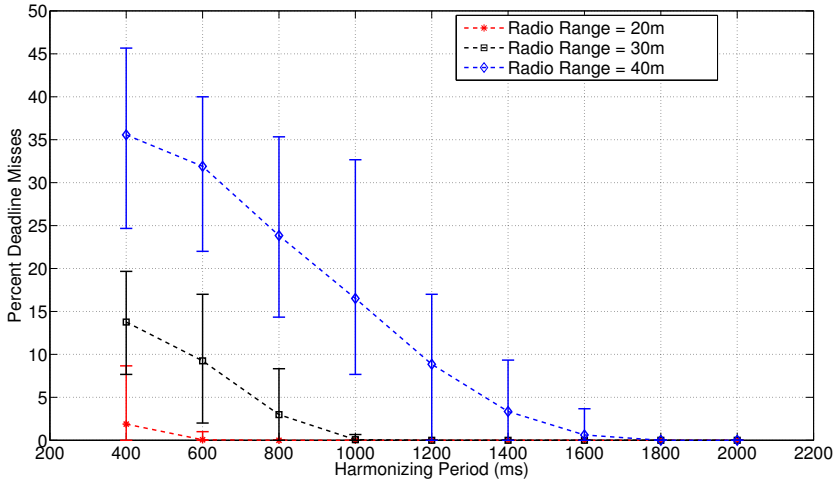


Figure 7.13: Impact on deadline misses with increase in T_H with different radio ranges, averaged over 20 iterations

transmit.

In this evaluation, we configured the transmission power of the nodes such that a receiver within a certain radius (in meters) can receive the packets successfully with a probability of 100%. Various external factors such as interference from other devices and multi-path can result in unexpected packet-loss, but our evaluation focuses on the performance limits of the protocol and its overall behavior under perfect packet reception. The simulation environment is chosen to highlight the advantages of the NHS protocol with respect to its real-time characteristics, and the impact of network capacity on deadlines.

We observed the impact of the network size and the selection of the harmonizing period on the real-time behavior of our protocol. The results are shown in Figure 7.13 and Figure 7.14. Firstly, we measured the effect of the choice of harmonizing period on the number of packets that miss the minimum deadline. The deadline is chosen corresponding to the harmonizing period as given by (7.7). The harmonizing period is defined according to the minimum possible period (and deadline) to show the performance limits of the NHS protocol. Longer deadlines are bound to provide better performance in terms of deadline misses by allowing the choice of larger T_H. Smaller harmonizing periods will require more packets to be transmitted within a smaller window, thus more deadlines will be missed. In Figure 7.13, we show the decrease in deadline misses as the harmonizing period

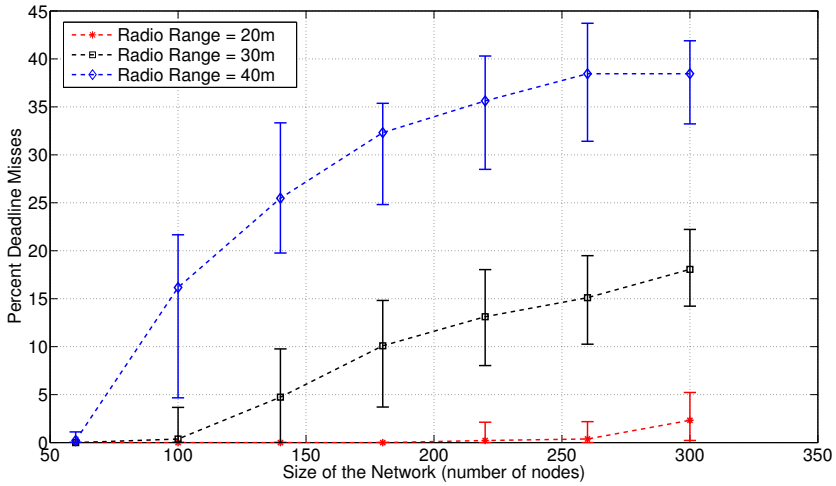


Figure 7.14: Average deadline misses over 20 iterations with respect to the size of the network. The error bars are also shown.

increases, for different values of the radio range. The total number of nodes, N, was fixed to 100 for these experiments, and we assume that each node releases a packet every harmonizing period. Note that, with larger radio range, more nodes are covered within one broadcast domain and more slots are required at each hop. Hence, a short harmonizing period may not be sufficient to accommodate all the transmissions. For example, a period of $1000ms$ or larger is enough to guarantee that all network deadlines are met, if the radius of coverage by each node is equal to or less than $30m$.

The next experiment studied the impact on deadline misses as the number of nodes in the network increases, for different radio ranges, and a given harmonizing period of $1000ms$. As the number of nodes increases, more packets need to be transmitted in each slot, thus the number of deadline misses also increases, as shown in Figure 7.14. This figure also shows that, with an harmonizing period of $1000ms$, up to 100 nodes are supported with no deadline misses.

The energy consumption of a sensor node is directly dependent on the duty-cycle of the radio, and we measured the average duty-cycle over all the nodes for different values of the harmonizing period. If the harmonizing period can be made large while meeting the deadlines, then increasing the harmonizing period improves the duty-cycle. Note that the offline guarantees offered by NHS (presented in section 7.5) enable us check if our selection of the harmonizing period will allow us to meet all deadlines. One of the key advantages of NHS lies in the fact that the nodes are autonomously

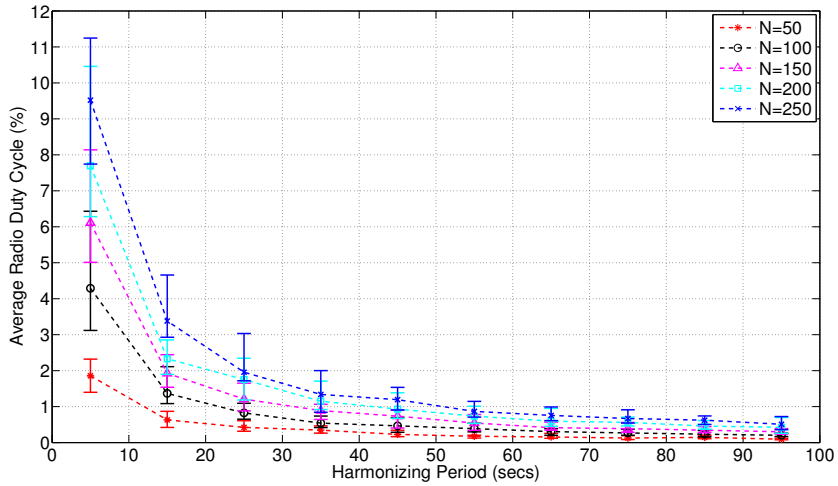


Figure 7.15: Average radio-duty cycle over all the nodes after 20 iterations with the increase in the harmonizing period.

assigned slots in the bootstrapping phase, and then the nodes do not need to listen to activity other than in the slots of its neighbors. A node only transmits in one slot per harmonizing period, and keeps the radio on during the transmission by its neighbors, which is always less than Q slots. The results for the average radio duty-cycle over all the nodes in the network with the varying network size are shown in Figure 7.15. The values are averaged over 20 iterations, and the error bars show the range of deviation in the duty-cycle. With a network size of 100 nodes, NHS can achieve about 0.50% duty-cycle at a period of 60 secs. The average duty-cycle remains below 2% for periods greater than 20 secs with a network size up to 200 nodes.

7.7.2 Latency and Throughput

We observed the end-to-end latency of a packet generated at different hop-levels with the help of simulated experiments. The results obtained are shown in Figure 7.16, where the time a packet is received at the root is shown on the y-axis. The time-values are scaled with respect to the harmonizing period, to highlight the relative delay in the reception of packets at the root node. Latency for different values of the cadence factor ω were also measured. The bold lines show the average latency for packets from different hop levels and the error margins show the maximum and minimum latency values observed in the experiments. The variability in the latency comes into play only because of the

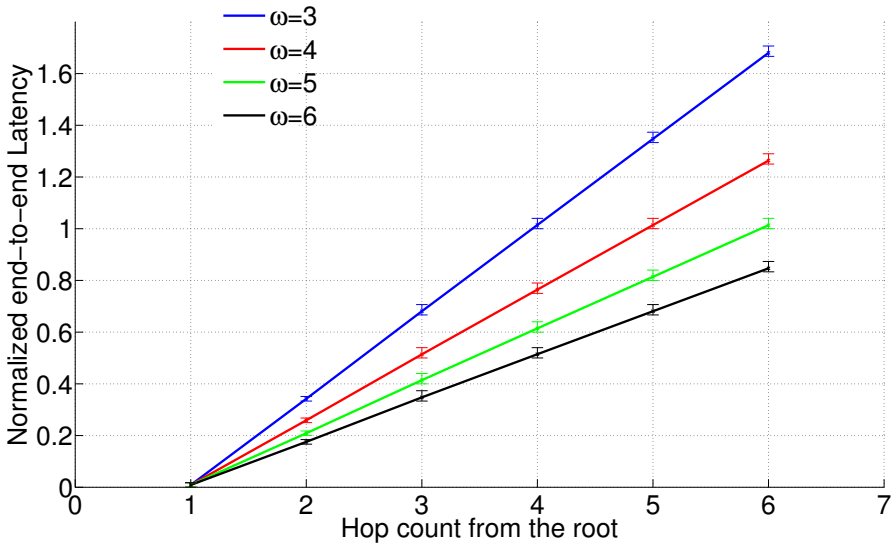


Figure 7.16: Average latency suffered by packets transmitted by nodes at different hop-levels. The x-axis correspond to the number of hops. The y-axis corresponds to latency in time-units normalized with respect to the harmonizing period.

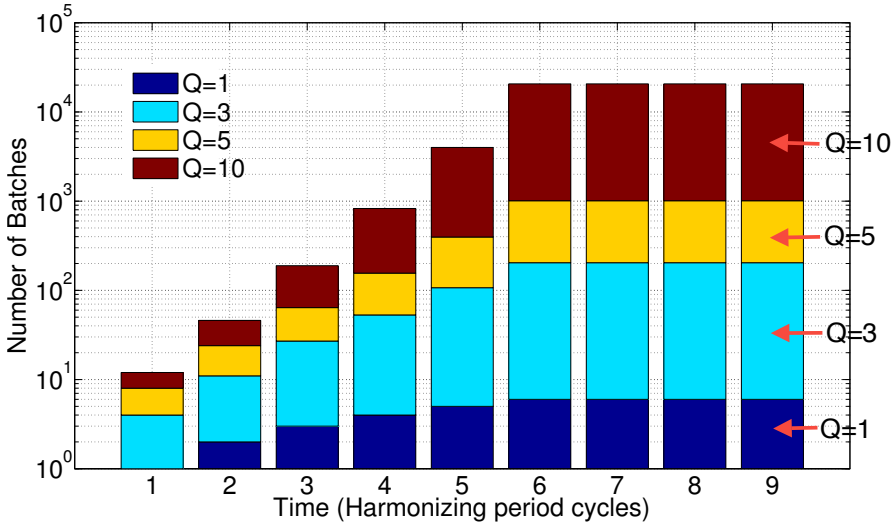


Figure 7.17: The number of packets received by the node with respect to time, shown on a logarithmic scale. Different plots correspond to different settings of neighbor degree Q.

relative location of the slot chosen by a node in its hop-level, which is small compared to the overall time-scale of the NHS operation. As the maximum number of hops in the network is 6, for $\omega = 6$, the maximum latency is less than the harmonizing period. For smaller values of ω, the packet delivery happens in the next cycle.

We also measured the number of packets received at the root node with respect to the maximum node-degree Q, which is shown in Figure 7.17. We chose different values of Q, namely: $1, 3, 5, 10$, which capture the overall density of the network. As expected, the number of received packets increases exponentially with time, until the protocol reaches steady state at the cycle number corresponding to the number of hops in the network. Once the steady state is reached, the root periodically receives the same number of packets indefinitely, if the network operation is not interrupted due to external events. The number of packets grow linearly for $Q = 1$, since in this case, the network is effectively a linear multi-hop topology.

7.8 Discussion

In this section, we discuss in detail the assumptions and limitations of the Network-Harmonized Scheduling protocol as presented. The NHS protocol assigns static transmission slots to devices in a network, where once a node chooses a slot in the bootstrapping phase, it uses the same slot (potentially) indefinitely. In this manner, the NHS approach is mainly suitable for static deployments with few or negligible topology changes. Typical examples of such deployments can include building monitoring, industrial sensing and other applications involving stationary sensor placements. NHS is a TDMA-scheduling approach that assigns static schedules. Hence, support for mobile nodes is not provided in the current version of the protocol. It is, however, possible to provide contention slots at each hop to allow mobile nodes to join the network and to facilitate existing nodes to request a new slot.

It must be noted that there are cases where the network topology obtained as a consequence of TDMA scheduling using the NHS protocol can be improved further. In particular, there is a possibility for two of more devices to choose the same slot where simultaneous transmissions can result in collisions. One such case is illustrated in Figure 7.18, where devices e and f with different parent nodes (a and b, respectively) may choose the same NHS slot and transmit simultaneously. This is

a case similar to the one shown in the timeline for multihop NHS in Figure 7.6. However, if nodes *e* and *f* lie within the communication domain, but are not connected to the same parent, collisions will occur when they transmit simultaneously. The bold-dashed line in Figure 7.18 shows such a communication link between *e* and *f*.

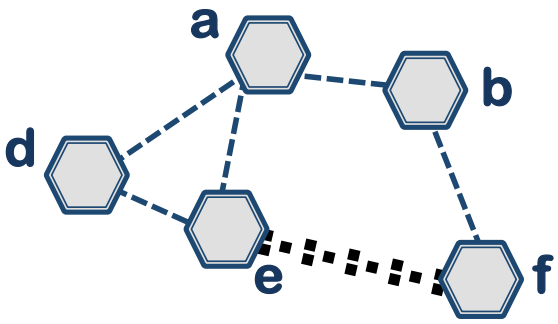


Figure 7.18: An example topology where nodes at same hop level (*e* and *f*) have different parents, *a* and *b*, respectively, but lie within their wireless communication range (shown with the thick dotted line). In the NHS protocol, nodes *e* and *f* may transmit simultaneously and can result in packet collisions.

The NHS protocol does not avoid the possibility of collision in such cases, as an explicit and optimum topology is not always achieved during the network setup (bootstrapping) phase in the current version of the protocol. Topology control [136] is typically a separate mechanism that generates an optimal subgraph from the overall connectivity graph in a network. Yao Graphs [137], Minimum Spanning Tree [138] and XTC [139] are some of the approaches that have been proposed in the past for topology control in ad-hoc wireless networks. To avoid the collision problem described above, we can use an approach similar to XTC, where, the NHS protocol can be modified to include a 2-pass mechanism. In the bootstrapping phase, nodes can share with their neighbors the list of nodes they are able to listen to, and also announce in advance the slot they will choose from the next cycle onwards. Once this information is shared, nodes can re-align their schedule if there is a chance of potential collision. With such a mechanism, node *f* can choose a different non-overlapping slot after listening to the announcement from node *e* about its transmission schedule for the next-cycle. We now provide a more detailed description of this mechanism.

Let us consider the example topology shown in Figure 7.18, where devices *e* and *f* can communi-

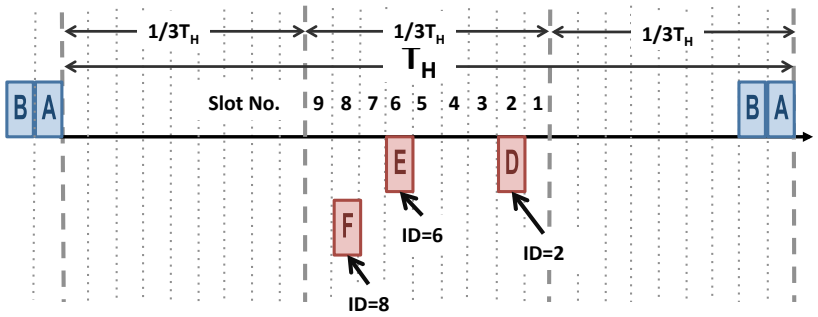


Figure 7.19: First cycle in the bootstrapping phase, where nodes choose slots corresponding to their unique id's (for the example shown in Figure 7.18).

cate with each other, but are connected to different parent nodes. As explained earlier in Section 7.3 and Section 7.4, the NHS protocol involves a bootstrapping phase where devices first choose unique slots corresponding to their id's. Therefore, nodes d, e and f choose unique slots as shown in Figure 7.19. In this cycle, all the nodes also listen to the medium for any other packets and hence, they become aware of their neighboring nodes at the same hop-level. In this way, node e can listen to the transmissions from nodes d and f, and node f can listen only to node e. As it is evident from the topology that node f is not aware of node d, it is not possible for node f to know which slot node e will choose in the next cycle for compressing the transmission schedule. To avoid this problem, we can include a second pass in the protocol as shown in Figure 7.20, where, instead of compressing the schedule in the second bootstrapping cycle, the nodes announce the slots they are going to choose from the third cycle onwards. These announcements are made in the same slots corresponding to the unique id's of the nodes, as in the first cycle. For example, node e will announce (in slot 6) that it is going to choose slot 2 from the third cycle onwards because it knows about only one node (node d) with a smaller id than its own. And node f will also announce (in slot 8) that it will choose slot 2 because it can listen to only one node (node e) with a smaller id than its own.

However, instead of transmitting directly in the slot announced by a node in the second cycle, each node also listens to the slots announced by the other nodes. In this way, nodes can come to know if there is a possibility for collision because of potentially overlapping slots. This conflict can then be resolved in the next (third) cycle, where a node with a larger id can choose the next higher slot instead of the slot it announced in the second cycle. Hence, node f can choose slot 3 instead of 2 as shown

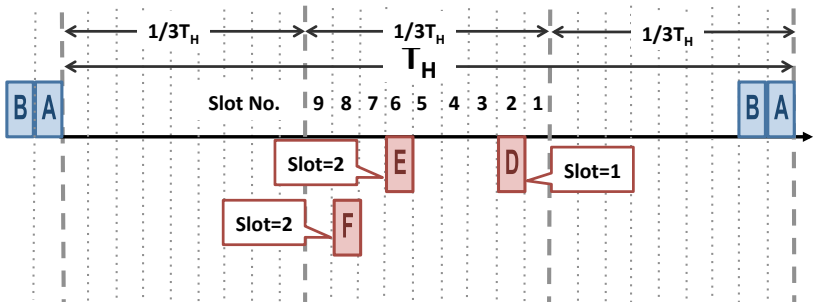


Figure 7.20: Second cycle in the bootstrapping phase, where nodes announce the slots they will choose from the next cycle onwards

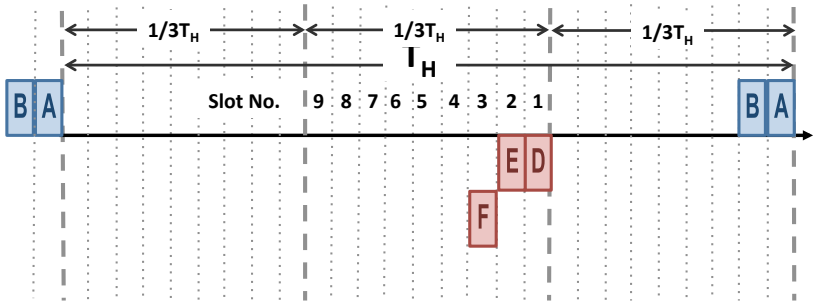


Figure 7.21: Third cycle in the bootstrapping phase, where devices choose slots such that potentially colliding slots can be avoided. Node *f* chooses slot 3 instead of slot 2.

in Figure 7.21. If there are multiple such devices with a potential for collision, they autonomously sort themselves in increasing order of their id's to choose transmission slots. Hence, by including one more cycle in the bootstrapping phase, where devices announce their transmissions slots for the next cycle, potential collisions can be avoided. Such a scheme for avoiding collisions needs to be evaluated in the future.

If there is still a possibility of collision between nodes associated with different parents, then the nodes can make use of an acknowledgement mechanism to identify their packet-delivery rate. In case of a low packet-delivery rate, a node can request a new slot to be assigned to it to avoid any possible collisions. For such a mechanism to converge to a schedule with no collisions, it might be beneficial to assign non-uniform number of slots at different hop-levels. Devices further away from the root with respect to the hop-distance can have a larger number of slots to choose from to avoid collisions. In general, the process of assigning slots to devices in a multi-hop network is

based on finding a *tree*-based topology, and the possibility of collisions in a network is dictated by the overall *connectivity-graph*. Finding a tree structure in the network graph to assign communication slots requires the knowledge of the entire network and is typically a centralized approach. Our NHS protocol, on the other hand, is a decentralized approach that assigns slots with only local information.

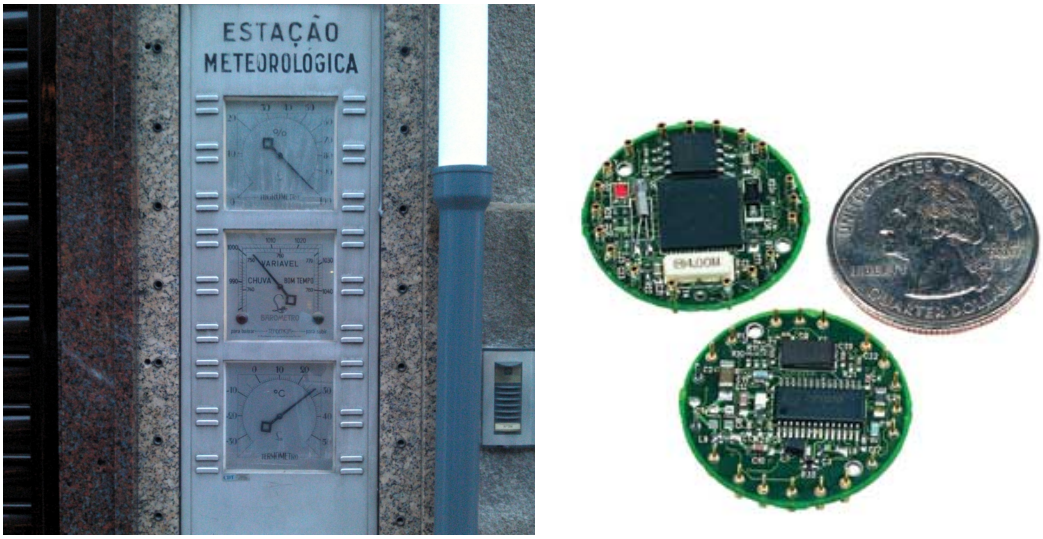
7.9 Summary

We proposed the Network-Harmonized Scheduling (NHS) protocol for distributed coordination of packet transmissions in a multi-hop network. The concept of NHS is inspired by the Rate-Harmonized Scheduling approach where the executions of various tasks are aligned around a period boundary for saving power by enabling the processor to go into deep sleep states more often. We use a similar approach to *batch* packets from multiple applications together around periodic boundaries, which makes the transmissions periodic. This periodic behavior is leveraged to create a network protocol that obviates the need for an explicit medium access protocol, and pipelines the packet transmissions over a multi-hop network. Our work shows that it is possible, and beneficial at the same time, to coordinate network access across multiple hops in a simple manner, without global state maintenance. This approach results in deterministic network operation, and allows offline delay guarantees to be derived for the protocol.

Chapter 8

Conclusions and Future Research Directions

The domain of Wireless Sensor Networks (WSNs) has seen a great amount of research beginning primarily from the concept of Smart-Dust consisting of tiny sensors scattered randomly in the field. The Smart-Dust vision is yet to accomplished, but a large number of projects covering a wide spectrum of problems and issues hass been successfully proposed, tested and implemented. The process of sensing the physical environment has evolved from large monitoring stations like the one shown in Figure 8.1a to tiny sensor nodes that are only few centimeter in size as the one shown in Figure 8.1b. Similarly, indoor-environment control has already evolved from simple analog devices like manually controlled thermostats to smart devices like the Nest [140] thermostats that can already interact with other home appliances such as washing machines as shown in Figure 8.2. The number of such interconnected devices that can sense the physical environment on one hand and accomplish actuation on the other hand, is bound to increase at a rapid pace. One of the biggest challenges for extracting the potential of a large number of devices interacting with each other is the development of an *ecosystem* of distributed applications. It is therefore plausible to argue that suitable application development/distribution platforms [141] for networked embedded systems can revolutionize the synergy of daily life operations with devices around us, much like the way Mobile Apps have revolutionized how we use mobile phones today.



(a) A weather sensing station in the downtown area of Porto city in Portugal.

(b) Coin sized sensor motes (Image source: Crossbow Technologies)

Figure 8.1: Sensing equipments: from stationary sensing stations on the left sensing temperature, pressure (barometer) and humidity (hygrometer) to tiny sensor nodes with wireless communication capability.

With this vision to enrich the ecosystem of applications for networked embedded systems and wireless sensor networks in particular, supporting multiple simultaneous applications is important. In this manner, a given infrastructure of embedded systems including sensors, actuators and other appliances can be more cost-effectively utilized by more than one independent user. This means that sensor networks should evolve from a typical application-specific technology to a multi-purpose infrastructure that, in addition to helping, should encourage users from varied backgrounds to develop and deploy their own applications.

In this dissertation, we developed several techniques that facilitate the development, execution and optimization of multiple applications on a sensor network infrastructure, with the overarching goal of making wireless sensor networks a popular technology among multiple users. Towards this goal, we proposed and implemented a holistic programming framework that allows users to develop network-level applications with little effort and also to re-program the network with ease. In addition to the programming support provided by our framework, we proposed relevant optimizations for reducing the overall resource usage on the sensor node because of multiple applications, and we also

Figure 8.2: A home appliance like a washing machine that can interact with a smart thermostat like the Nest. (Image source: Whirlpool Home Appliances website)

proposed an efficient networking protocol to harmonize the packet exchange a in multi-hop network.

8.1 Research Contributions

The research contributions of this dissertation are as follows:

- The overarching goal of this dissertation is the design of a multi-dimensional framework to support multiple applications on a sensor network infrastructure. It will help in further promoting the adoption of sensor networks in practical life such that independent users from diverse technical backgrounds can create multi-disciplinary applications.
- We proposed a framework called Nano-CF [34] to support multiple applications on a sensor network, such that network-level applications can be easily programmed and deployed. We also proposed a programming pattern called *sMapReduce* to make it easy to design sensor networking applications by splitting them into *sMap* and *Reduce* functions.
- We proposed optimizations that identify and eliminate various sources of over-consumption of resources in a sensor network in the case of multiple applications. This is achieved by elimi-

nating redundancy across applications [37] at the level of an individual node. Moreover, a hierarchical assignment scheme is proposed that meets the resource constraints on a sensor node, while maximizing the redundancy elimination across applications.

- We also proposed Network-Harmonized Scheduling approach that batches packet transmissions from multiple applications and harmonizes the network operation [42] to achieve TDMA-like energy efficiency.

- This work contributed to the Nano-RK source-code repository by adding the Nano-CF programming framework to it and incorporating Network-Harmonized Scheduling in the Contiki operating system as a link- and transport layer protocol.

We now provide a more detailed description of these contributions.

8.1.1 Programming framework for supporting multiple applications

In this dissertation, we presented a macro-programming framework to support more than one application on sensor networks. Our proposed framework is called Nano Coordination Framework (Nano-CF) [34] and it is built on top of the Nano-RK resource kernel. Nano-CF includes a compiler, application manager, and runtime layer to help the development, deployment and executions of applications, respectively. The applications are created using our Nano-CL programming language with descriptors that can be used to specify a set of nodes to execute a given application.

With Nano-CF, we demonstrated that supporting multiple applications is possible and advantageous even on resource-constrained sensor nodes, and by making use of the resource kernel properties of Nano-RK the isolation among applications and the fair usage of resources can be enforced. We showed using representative examples that simple data collection applications can be developed with only 2-5 lines of code in Nano-CL, and a slightly more complex application aiming occupancy monitoring requires only about 20 lines of code. The overhead incurred due to the execution of applications in the form of interpreted bytecode is qualitatively offset by the ease of programming in terms of the number of lines of code.

8.1.2 Compile-time Inter-application Redundancy Elimination and Hierarchical Assignment

With multiple applications executing on a sensor node, redundant sampling of sensors can occur and lead to over consumption of resources. To eliminate this redundant sampling, we proposed a scheme called Redundancy Elimination with Implicit Scheduling (REIS) [37], that identifies the overlap (redundant sensor access) across applications by making use of the well-known string matching algorithms, namely, Longest Common Subsequence and Shortest Common Supersequence. The REIS approach creates a monolithic block from all the input applications such that the independent applications are logically executed, but at the runtime level only one monolithic application executes. We showed that this scheme can save resources and energy consumed by the processor on a sensor network. We also introduced a hierarchical assignment scheme [142] where instead of creating a monolithic block from all the input applications, we created a set of task-blocks that execute independently which helps in resource management while removing redundancies.

8.1.3 Network-Harmonization to coordinate packets transmitted by multiple applications

We proposed a network protocol called Network-Harmonized Scheduling (NHS) [42] that coordinates packets across a multihop network such that the packet transmissions are harmonized along periodic boundaries. This makes the network behavior more deterministic even though the nodes choose their slots autonomously with only neighborhood information and the logical topology is also created on the fly without the need for a central coordinator. At the node level, NHS harmonizes the packet releases from multiple applications along periodic boundaries, such that the overall packet release into the network can be made periodic. NHS leverages this concept to achieve network-level coordination. We showed that NHS can achieve energy efficiency close to that of an ideal TDMA protocol in terms of energy consumption.

8.2 Validation of the Thesis Statement

We now revisit the thesis statement presented earlier in Chapter 1 and we validate it based on the studies undertaken as a part of this dissertation.

> *A holistic framework with suitable optimizations can be designed for the deployment and execution of multiple applications on a sensor network infrastructure without compromising the resource usage and the overall energy consumption of the network.*

We proposed the Nano-CF programming framework that allows the development, deployment and management of multiple applications of a sensor network. Users can create network applications using the Nano-CL programming language with significant ease in a very small number of lines of code. The additional overhead in resource-usage because of the redundant sampling of sensors by different applications is eliminated using the Redundancy Elimination with Implicit Scheduling (REIS) approach. Finally, the overall network behavior is coordinated using Network-Harmonized Scheduling (NHS), that harmonizes the packet transmissions across the network while reducing the average radio-duty cycle at each node in the network.

8.3 Future Directions

This dissertation addresses several aspects regarding the support of multiple applications, but some challenges remain unaddressed that can be undertaken in the future to build our proposed solutions.

8.3.1 Multiple Applications on Dynamic Networks

The network topologies assumed in this dissertation are based on static, infrastructure-like sensor networks that involve nodes deployed at fixed locations. However, in several cases, new devices may join or leave the network quite frequently. Supporting multiple users with multiple applications in such a dynamic network is a challenge that may require designing novel techniques. An advertising mechanism may need to be designed so that a new device may join and broadcast its capabilities and receive not only a joining confirmation but also one or more applications to execute. In such a case, applications may also need to be re-assigned from the other nodes in the network because the newly

joined node and the network topology itself may change. For example, an application that collects the maximum luminosity levels in a room may benefit if executed on a new node placed closer to a bright window. In such a case, the other nodes that sent their sensor reading as the maximum value might only act as a router rather than a data source with the addition of the new better-placed node.

8.3.2 Distributed Responsibility sharing among sensor nodes

In our Nano-CF framework, the applications are assigned explicitly to all or a subset of nodes in a network. Designing distributed responsibility sharing algorithms can be beneficial in decoupling the application development from the application execution. A programmer may only need to specify the spatial scope of the application and it becomes the responsibility of the network to decide which node or a set of nodes is best to execute a particular application autonomously, while optimizing the overall resource usage. Considering building monitoring as an example, an application may only specify that it requires one temperature measurement from every room in a building and, in a deployment with more than one temperature sensor per room, the most efficient nodes may be chosen to send their sensor readings in a distributed way. One way to achieve this goal is by using a shared data structure that can be exchanged within a neighborhood of nodes. Such a data structure can include id's of the nodes, their location, the applications that can be supported by a particular node, the applications that a node is currently executing and several other attributes that can help in making distributed optimization possible. With regular exchange of this data structure, the network can converge to a state where only the most suitable nodes execute a given application and, with a few iterations, the network converges to a stable application mapping. However, the challenges in this approach is to ensure that the application requirements are met at all times, even during the convergence, and that the size of this data structure does not become prohibitively large.

8.3.3 A General Application Model

In this dissertation, we have assumed a strictly periodic model for applications, where the periods are known at design time. This model, however, does not cover all the possible types of applications in sensor networks. Several applications may be an event triggered, where certain external factors may start their execution. A simple example could be the case where, based on the value of a sensor or the

reception of a sporadic message, an application is executed. On one hand, periodic applications can cover a large variety of applications and can model event-based behavior to some extent, but on the other hand, handling more generic application types by design can allow more fine-tuned resource optimizations.

8.3.4 Context-aware application deployment

One of the challenges expected to be faced by the next-generation technologies such as smart-home or building automation is context-aware applications. Context awareness typically refers to leveraging the knowledge of several attributes about a user such as location, neighborhood and ambient environment, to provide smarter services. A typical context-aware system has the potential to become much more flexible if it has the provisions to deploy and execute new collaborative and distributed applications at runtime. Such a system should support in an easy manner the definition of more than one context and a corresponding action to be taken by a set of devices. Our framework could be extended with support for multiple contexts, context resolution and the ability to take resulting actions.

8.4 Prospective Vision: A Co-Operating System

In conclusion and as a future research direction, we present a vision and a design proposal for a new paradigm of an operating system for networked embedded systems which can envelope the optimizations proposed in this thesis, further facilitate the development of distributed applications and tackle the unaddressed issues mentioned in the previous section.

The increase in penetration of embedded-systems has led researchers in academia and industry to envision an *Internet of Things*, where many objects in the physical world will be connected to each other and the Internet. Such a highly connected world will further enable several applications like home automation, intelligent ambience, green buildings and so on. However, the full potential of highly-connected cooperating objects is still difficult to perceive, as there is scope for diverse and revolutionary applications that may not have been conceived yet.

To enable the development of such new applications, new paradigms for embedded systems

software are required. We believe that the currently available operating systems and programming abstractions may not encourage an environment for active application development for future networked embedded systems. We argue that the design of the operating systems for networked embedded systems needs to be thought from a different perspective than the one already taken in popular solutions like TinyOS [56], Contiki [57] or Nano-RK [35]. Most of the popular research studies in the direction of facilitating programming on sensor networks assume that the existing operating systems are the *de-facto* platforms upon which the middleware or the programming abstractions have to be built. This assumption needs to be thought again from a top-down perspective where the new goal is to support dynamic deployment and management for network-level applications.

Existing operating systems were designed to ease the programming of specific hardware that was developed as prototypes for wireless sensor networks. Programming these devices on *bare metal* is complex and requires a high degree of expertise in embedded systems. Platforms like MicaZ and TelosB are resource-constrained yet powerful-enough devices that can easily support a small operating system, custom communication stacks and one or more applications. Operating systems were designed from the perspective of easing the application development process on individual devices because, even in their standalone operation, they are complex systems with a processor, a radio, several sensors, a programming/communication interface over the USB or the serial port and so on. These hardware and software platforms have contributed a lot towards the development of groundbreaking research and proof-of-concept ideas. Moreover, the research in these areas provided a vision for the future of networked embedded systems. To achieve the goal of ubiquitous connectivity of embedded devices described earlier, there is a need to design (distributed) operating systems from scratch that completely isolate the users from node-level intricacies, and take the application development to a higher level where the whole network ecosystem can be viewed as a single system. We believe that revamping the way operating systems are designed is a first step towards this goal.

By networked embedded systems, we refer to the broader area of Cyber-Physical Systems (CPS) that react to the environment in addition to just sensing the physical quantities as in the case of wireless sensor networks. Timeliness is an important requirement of CPS, because of the tight integration of sensing and actuation. We believe that it is time we move from an operating system to a *co-operating system* or *CoS*, that embodies all fundamental functionalities necessary for encouraging application development for networked embedded systems directly above it. *CoS* is a truly distributed operating

system, in the way that it provides a geographically distributed view of the operating system to the user rather than abstracting the network as a single machine. In the rest of this chapter, we describe a few key principles that can motivate the design of such a co-operating-system, and we propose a possible architecture that can satisfy some of those principles.

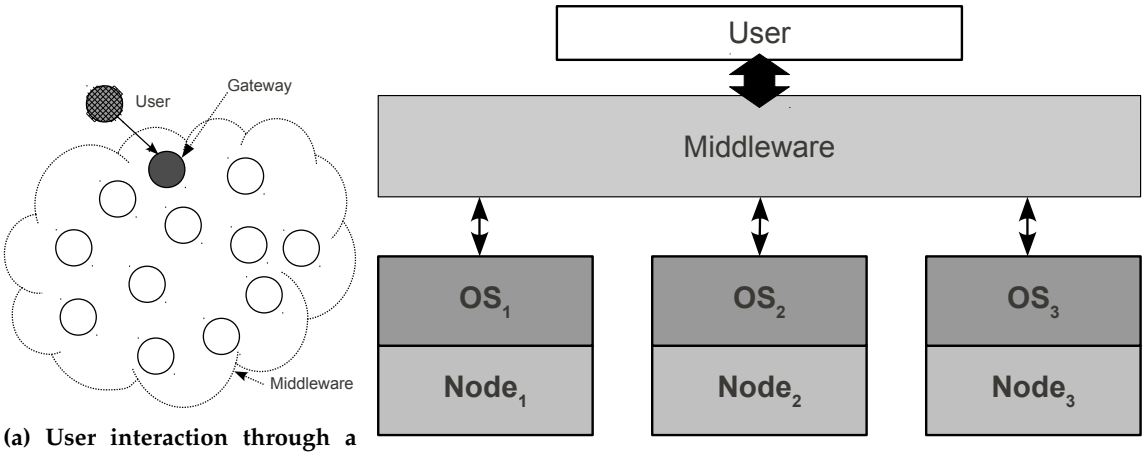
8.4.1 Design Principles behind a CoOperating System

The motivation behind *CoS* is driven by the observation that the existing operating systems were only designed for facilitating node-level application development, and middleware solutions designed to allow *macro-* or network-level programming are limited in scope and are highly dependent on the underlying operating systems. We discuss some of the principles that stress on the need for a new perspective in the design of an operating system for cyber-physical systems.

Programming using *CoS*

As we have outlined in this dissertation, traditional network programming approaches typically involve a middleware layer or a programming abstraction that inevitably tends to centralize the network topology. A user has to interact with a programming layer, that generally resides on a central server or a gateway. As shown in Figure 8.3, the middleware abstracts the network complexities from a user with the help of a hierarchical setup that can be rigid and highly application-specific.

In contrast, *CoS* is designed to facilitate application development in a distributed way for networked objects of the future. The programmer interacts with the operating system directly for creating network-level applications, instead of a middleware or a gateway. *CoS* makes it possible that the user can interact with the system at any logical or topological location in the network, as shown in Figure 8.4a. *CoS* manages the communication and dissemination of the application program to other nodes, based on the user requirements embedded in the logic of the application. The communication between the nodes can be transparent to the user, and the interaction among them is dictated by the application logic. The user may deploy an application over one or more nodes, each running an instance of *CoS*. Then the application is distributed to other participating nodes by *CoS* as exemplified in Figure 8.4b.



(a) User interaction through a middleware

(b) Layered diagram showing the user interaction

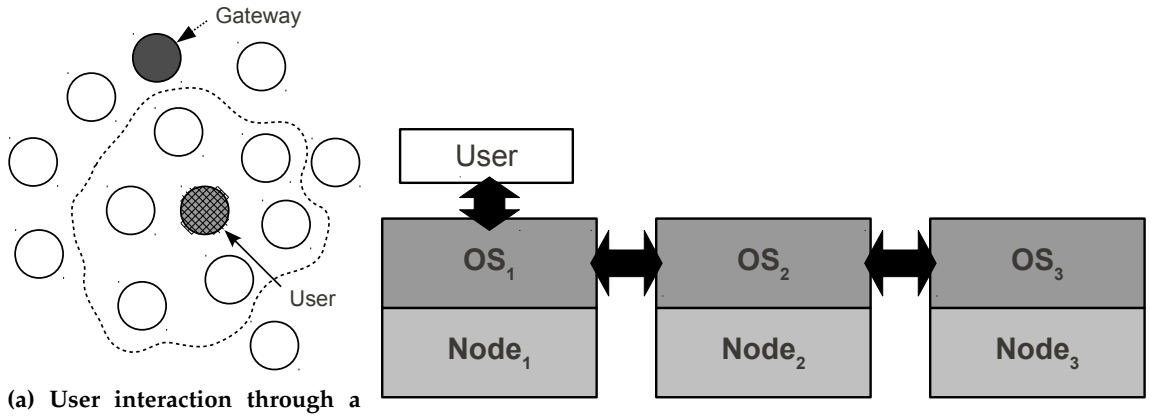
Figure 8.3: Programming with the help of a middleware, to emphasize the centralizing aspect

Truly Distributed Design

Traditional distributed operating systems were designed from a perspective of abstracting away the presence of more than one machine from the user. According to Tanenbaum *et al.* in [143]:

> *"As a rule of thumb, if you can tell which computer you are using, you are not using a distributed system. The users of a true distributed system should not know (or care) on which machine (or machines) their programs are running, where their files are stored, and so on."*

This presents a major disconnect from cyber-physical systems, where the devices are not only logically distributed but geographically as well, and it is often important for the applications to associate the geographical location in their logic. For example, controlling the window blinds based on light level readings in specific rooms *A, B and C*. Most middleware and programming abstractions tend to centralize the network, and cause overheads in maintaining connectivity to all the nodes and may also involve participation of the nodes that may otherwise not be required. A middleware layer would require a hierarchical architecture to enable the deployment of applications. Similar to the example of vehicles at a junction, the nodes may not provide enough support for an active higher layer in dynamic topologies. Hence, the operating system executing on the nodes needs to allow application deployment in a decentralized way, and then execute distributed logic. A distributed operating system for cyber-physical systems (logically and geographically) can decentralize the operation of the



(a) User interaction through a *CoS*

(b) Layered diagram showing application dissemination using *CoS*

Figure 8.4: Programming using a *CoS*

network. It should allow a user to deploy applications by connecting to any one or more nodes in the network. *CoS* should obviate the requirement of having a middleware layer and/or a programming abstraction to program the network.

Other Key Features

Modular: *CoS* should be modular in design supporting dynamic loading and unloading of modules and applications. This can help a programmer create powerful applications with significant ease by making use of existing modules, or creating new ones. Modules can either reside on the system flash memory, or can be delivered over the air, if needed.

Integrated Network Management: *CoS* should support tightly integrated network management, such that a device is able to discover its neighboring devices and thus allowing the updating of routing information. This information should be made available to the programming interface for network-wide application development.

Programming Interface: The traditional way of programming sensor nodes via a direct one-to-one connection to a computer is designed to facilitate application development while making good use of on-board peripherals. The programming interface of the proposed operating system should be at the network-level by design, rather than being a layer on top of node-level programming abstraction. The programming interface should have information about the network topology and capability of

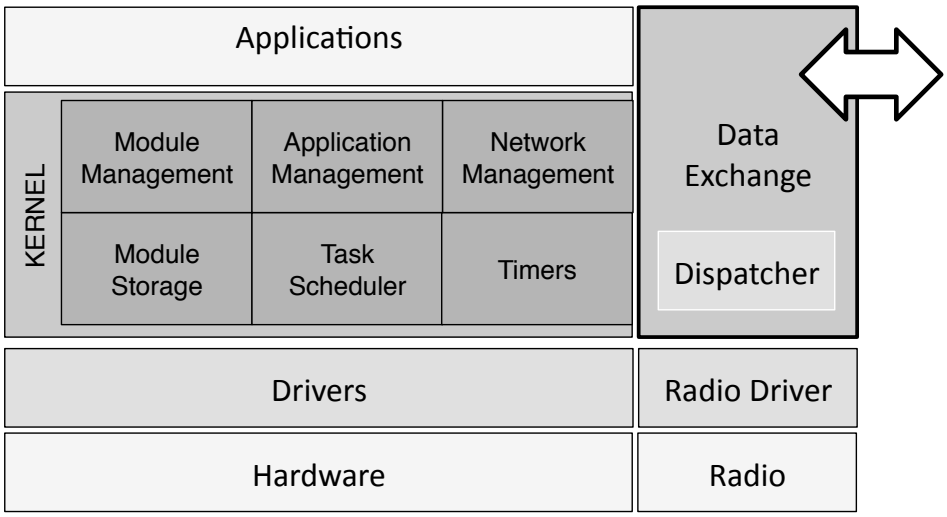


Figure 8.5: Typical architecture of a *CoS*

the nodes to provide a global view to the programmer in an intuitive way.

Isolation of Applications: Multiple independent applications on a network of nodes should be supported, while making sure that the data exchange and operation of the applications remains isolated. The data may need to be multiplexed in the network to save energy, and then de-multiplexed to deliver to each user.

Support for Heterogenous Platforms: The real-world applications of sensor networks, especially in the context of cyber-physical systems, may require diverse hardware, including processor, sensors, actuators and communication peripherals. The operating system should be designed such that it supports this varied ecosystem of hardware, and provides suitable programming provisions.

8.4.2 *CoS* Architecture

After discussing the design features of *CoS*, we now propose its architecture. An outline of the architecture showing the various components is provided in Figure 8.5. The following important components constitute *CoS*: Kernel, drivers, a data exchange plane and the applications. We will briefly describe each of those next.

Applications

One of the key motivating principles behind the design of *CoS* is easy and convenient deployment of applications directly on top of the *operating system*, rather than having a network-wide middleware or a programming framework. The *status quo* in the programming of networked embedded systems involve either copying operating system images with embedded applications onto to the flash, or using a network-level virtual-machine delivery system. Following from the typical trend of writing applications on top of an OS for general-purpose computing systems, *CoS* should allow installing applications at runtime, without the need of a middleware layer. Given the resource-constrained nature of embedded systems, a programmer can create and compile applications on a PC, and then deliver the binaries to *CoS*.

As explained earlier, the user may not need to depend on a gateway or a central-server to deploy the application, *CoS* allows any one or more nodes to act as the *point of delivery*. The distribution of an application to participating nodes is handled by the data exchange plane. The kernel provides a *mount point* to the newly deployed application and adds its information to a local list. The mount-point is a pointer to the memory location where the application resides, so that it can be executed according to the scheduling policy of *CoS*, and the criticality or the priority of the application.

Each application has to specify a scope, both geographic and logical, to help determine the nodes to be associated with that application. In case of more than one application submitted by independent users, *CoS* ensures isolation between the state of the applications both at the node- and the network-level, thus making sure that the data is delivered to the intended destination in a seamless manner and appropriate action is taken in case of *sense-and-react* applications. This may require conflict resolution or deadlock avoidance support from the kernel. For example, in an intelligent surroundings scenario, if one application requires lights off in the night, and another requires the lights to be turned on if the window shades are down, it may happen that they can be in conflict at some point in time. This conflict has to be resolved among participating applications, and this responsibility lies with the kernel to take a default action while taking applications semantics into account.

Kernel

The kernel handles the core functionality, including managing the applications, task scheduling and timing. Scheduling the applications is one of the most important functions of the kernel. The underlying hardware platform may be significantly resource-constrained that may not support more than a certain number of applications, and the resource usage of applications has to be limited within certain bounds. The kernel ensures that the applications do not misbehave, and their timing requirements are met in the best manner possible. For this purpose, the kernel from the Nano-RK [35] operating system can be adapted, as it has support for real-time scheduling and task-level resource reservations. In addition to these optimizations, the kernel should be able to resolve conflicts in case of dissimilar requirements of applications. The kernel can make use of priorities, or assign default behaviors to the peripherals. In the example of conflict in the state of lights mentioned earlier, the kernel may choose to keep the lights off at night, until over-ridden manually.

The kernel should be modular in design so that drivers or other modules can be added to enable required functionalities. Cyber-physical systems can consist of varied sensor and actuator peripherals, and providing out-of-the-box support for such possibly large number of devices may not be practical. Programmers or users should be able to install modules on the nodes covered by their applications. The kernel should allow dynamic loading and unloading of modules in a manner similar to the SOS [144] operating system. The kernel can achieve this with the help of module management and storage components.

As *CoS* may be run on battery-powered devices, minimizing the power consumption is important. A power management module tries to put the device to sleep for as long as possible. Nodes may operate at very low duty-cycles, hence the power-management module can ensure that different applications execute in way to maximize the sleep interval.

Drivers

Hardware support for the peripherals on a node, including the radio, the sensors and the actuators, is provided through drivers. In addition to the default drivers available with *CoS*, drivers can be loaded as modules at runtime. Such a design allows the easy integration of heterogenous devices and dynamic behavior in the long-term. The operating system does not need to be *flashed* again if some

peripheral devices are added or removed. In addition to the peripherals, drivers can help applications to configure the communication layer as well. Radio configuration, medium-access control and routing can be implemented as modules and changed on the fly, if needed.

Data Exchange Plane

One of the most important components of the *CoS* architecture is the data exchange plane. The data exchange plane handles all the communication to and from the node. Applications created by the user are delivered to the nodes through this plane, and are further relayed to other nodes that participate in the given application. Other responsibilities of the data exchange plane are ensuring isolation between the applications, delivering data to the nodes involved, and also directing actuation based on the distributed logic of an application.

The data exchange plane uses information from the network management module in the kernel about the topology and routing information in order to maintain the communication across a multi-hop network. It can use a device-advertisement phase to construct a topology map of the system. The advertisements allow the exchange plane to maintain information about the capabilities of the neighboring nodes. The radius of the neighborhood may be pre-configured as a design-parameter or specified by the applications. Developing an application may require knowledge about the capabilities of the devices in the network and, hence, the advertisements available to the data exchange plane should be provided to the programmer so that a distributed logic can be implemented, in accordance with the truly distributed design principle explained in Section 8.4.1.

The flexibility of *CoS* lies mainly in the configurability of the data exchange plane and how conveniently a programmer can access and adapt this plane in her application. It allows on-demand information gathering about the devices around and topology formation according to the application needs. For more dynamic network topologies, the maintenance of network information and device advertisements can be more frequent if an application requires so. Otherwise, the network may remain relatively dormant if no application-level updates are required.

8.4.3 Summarizing the Co-operating System

We briefly discussed a potential research direction that consists of a new paradigm in operating system design called *Co-operating System* or *CoS*, that aims to ease the application development for cyber-physical systems. We argued that current sensor networking operating systems like TinyOS, Contiki or Nano-RK are designed with a goal to facilitate the programming of individual nodes in a network of embedded devices. Middleware or network programming frameworks are the other end of the spectrum that may reduce the flexibility of applications and jeopardize the reliability and robustness. Perhaps this is the reason that, even with the development of several such solutions, not many have been widely adopted, and researchers still depend heavily on developing applications directly on top of the operating system. We presented our design principles behind *CoS* and discussed its architectural aspects that may enable significant changes in the way applications are developed and distributed for networked embedded systems. It can be argued that *CoS* may not be significantly different from a middleware layer running on top of a traditional OS in terms of the software-architecture, but the fresh perspective of creating network applications directly on *CoS* can provide a conducive setup for rapid and diverse application development for cyber-physical systems.

In this dissertation, we presented our approach to facilitate and optimize multiple applications developed by independent users. In such scenarios, where different users can make use of a given infrastructure, security and privacy concerns can be of paramount importance and is a future research area to be studied.

Chapter 9

Bibliography

[1] Rowe A., Mangharam R., Rajkumar R., "FireFly: A Time Synchronized Real-Time Sensor Networking Platform," *Wireless Ad Hoc Networking: Personal-Area, Local-Area, and the Sensory-Area Networks, CRC Press Book Chapter*, 2006.

[2] Andrea Azzara, Daniele Alessandrelli, Stefano Bocchino, Matteo Petracca, and Paolo Pagano, "Pyot, a macroprogramming framework for the internet of things," in *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, June 2014, pp. 96–103.

[3] Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft, "Senshare: transforming sensor networks into multi-application sensing infrastructures," in *Proceedings of the 9th European conference on Wireless Sensor Networks*, Berlin, Heidelberg, 2012, EWSN'12, pp. 65–81, Springer-Verlag.

[4] Arsalan Tavakoli, Aman Kansal, and Suman Nath, "On-line sensing task optimization for shared sensors," in *IPSN '10: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, Stockholm, Sweden, 2010, pp. 47–57, ACM.

[5] M.T. Hansen, R. Jurdak, and B. Kusy, "Unified broadcast in sensor networks," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, 2011, pp. 306–317.

[6] Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal, "Desync: self-organizing desynchronization and tdma on wireless sensor networks," in *Proceedings of the 6th international confer-*

ence on Information processing in sensor networks, New York, NY, USA, 2007, IPSN '07, pp. 11–20, ACM.

[7] J. C. R. Licklider, "Memorandum For Members and Affiliates of the Intergalactic Computer Network," April 1963.

[8] Robert R. Schaller, "Moore's law: Past, present, and future," *IEEE Spectr.*, vol. 34, no. 6, pp. 52–59, June 1997.

[9] Montenegro G., Kushalnagar N., Hui J., Culler D., "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," *Internet Engineering Task Force RFC 4944*, 2007.

[10] J. C. R. Licklider, "Man-Computer Symbiosis," *IRE Transactions on Human Factors in Electronics*, vol. HFE-1, pp. 4–11, Mar. 1960.

[11] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th symposium on Operating systems design and implementation*, Seattle, Washington, 2006, OSDI '06, pp. 381–396, USENIX Association.

[12] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong, "A macroscope in the redwoods," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 51–63, ACM.

[13] I. Vasilescu, K. Kotay, D. Rus, M. Dunbabin, and P. Corke, "Data collection, storage, and retrieval with an underwater sensor network," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 154–165, ACM.

[14] J. Heidemann, Wei Ye, J. Wills, A Syed, and Yuan Li, "Research challenges and applications for underwater sensor networking," in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, April 2006, vol. 1, pp. 228–235.

[15] K. Chintalapudi, T. Fu, Jeongyeup Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri, "Monitoring civil structures with a wireless sensor network," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 26–34, March 2006.

[16] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon, "Wireless sensor networks for structural health monitoring," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, New York, NY, USA, 2006, SenSys '06, pp. 427–428, ACM.

[17] Mo Li and Yunhao Liu, "Underground structure monitoring with wireless sensor networks," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, New York, NY, USA, 2007, IPSN '07, pp. 69–78, ACM.

[18] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis, "Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 64–75, ACM.

[19] V.C. Gungor and G.P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *Industrial Electronics, IEEE Transactions on*, vol. 56, no. 10, pp. 4258–4265, 2009.

[20] Ian Johnstone, James Nicholson, Babar Shehzad, and Jeff Slipp, "Experiences from a wireless sensor network deployment in a petroleum environment," in *Proceedings of the 2007 International Conference on Wireless Communications and Mobile Computing*, New York, NY, USA, 2007, IWCMC '07, pp. 382–387, ACM.

[21] Kok-Kiong Yap, Vikram Srinivasan, and Mehul Motani, "Max: Human-centric search of the physical world," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 166–179, ACM.

[22] Laibowitz M., Gips J., Aylward R., Pentland A., Paradiso J., "A Sensor Network for Social Dynamics," *International Conference on Information Processing in Sensor Networks (IPSN)*, 2006.

[23] C.R. Baker, K. Armijo, S. Belka, M. Benhabib, V. Bhargava, N. Burkhart, A Der Minassians, G. Dervisoglu, L. Gutnik, M.B. Haick, C. Ho, M. Koplow, J. Mangold, S. Robinson, M. Rosa, M. Schwartz, C. Sims, H. Stoffregen, A Waterbury, E.S. Leland, T. Pering, and P.K. Wright,

"Wireless sensor networks for home health care," in *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, May 2007, vol. 2, pp. 832–837.

[24] Liyang Yu, Neng Wang, and Xiaoqiao Meng, "Real-time forest fire detection with wireless sensor networks," in *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, Sept 2005, vol. 2, pp. 1214–1217.

[25] Jyh-How Huang, Saqib Amjad, and Shivakant Mishra, "Cenwits: A sensor-based loosely coupled search and rescue system using witnesses," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 180–191, ACM.

[26] F. Osterlind, E. Pramsten, D. Roberthson, J. Eriksson, N. Finne, and T. Voigt, "Integrating building automation systems and wireless sensor networks," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 1376–1379.

[27] Xiaofan Jiang, Minh Van Ly, Jay Taneja, Prabal Dutta, and David Culler, "Experiences with a high-fidelity wireless building energy auditing network," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2009, SenSys '09, pp. 113–126, ACM.

[28] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao, "Racnet: a high-fidelity data center sensing network," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2009, SenSys '09, pp. 15–28, ACM.

[29] Anthony Rowe, Mario Berges, Gaurav Bhatia, Ethan Goldman, Raj Rajkumar, Lucio Soibelman, James Garrett, and Jose Moura, "Sensor andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development: Special Issue on Smarter Cities and Sensed Infrastructures*, 2010.

[30] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, "Towards multi-purpose wireless sensor networks," in *Systems Communications, 2005. Proceedings*, aug. 2005, pp. 336 – 341.

[31] Ryan Newton, Greg Morrisett, and Matt Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th international conference on Information processing in sensor networks*, Cambridge, Massachusetts, USA, 2007, IPSN '07, pp. 489–498, ACM.

[32] Philip Levis and David Culler, "Matè: A tiny virtual machine for sensor networks," in *10th conference on Architectural support for programming languages and operating systems*, San Jose, California, 2002, ASPLOS-X, pp. 85–95, ACM.

[33] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[34] V. Gupta, Junsung Kim, A Pandya, K. Lakshmanan, R. Rajkumar, and E. Tovar, "Nano-cf: A coordination framework for macro-programming in wireless sensor networks," in *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*, June 2011, pp. 467–475.

[35] A. Eswaran, A. Rowe and R. Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks," *IEEE Real-Time Systems Symposium*, 2005.

[36] Vikram Gupta, Eduardo Tovar, Luis Miguel Pinho, Junsung Kim, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar, "smapreduce: A programming pattern for wireless sensor networks," in *Proceedings of the 2Nd Workshop on Software Engineering for Sensor Network Applications*, New York, NY, USA, 2011, SESENA '11, pp. 37–42, ACM.

[37] Vikram Gupta, Eduardo Tovar, Karthik Lakshmanan, and Raj Rajkumar, "Inter-application redundancy elimination in wireless sensor networks with compiler-assisted scheduling," in *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*. 2012, pp. 112–119, IEEE.

[38] Daniel S. Hirschberg, "Algorithms for the longest common subsequence problem," *J. ACM*, vol. 24, pp. 664–675, October 1977.

[39] Kari-Jouko Raiha and Esko Ukkonen, "The shortest common supersequence problem over binary alphabet is np-complete," *Theoretical Computer Science*, vol. 16, no. 2, pp. 187 – 198, 1981.

[40] Atmel, "Atmega 128rfa1 data sheet," 2011.

[41] Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar, "Rate-harmonized scheduling for saving energy," in *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, Barcelona, Spain, 2008, pp. 113–122, IEEE Computer Society.

[42] Vikram Gupta, Eduardo Tovar, Nuno Pereira, and Ragunathan (Raj) Rajkumar, "Network harmonized scheduling for multi-application sensor networks (to appear)," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, Aug 2014.

[43] Crossbow Technology, "Micaz sensor network platform datasheet," `http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`, 2013.

[44] Crossbow, "MICAz Datasheet ," `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf`, 2004.

[45] "Sun spots," `http://www.sunspotworld.com/`, Accessed: July 7, 2014.

[46] Joseph Polastre, Robert Szewczyk, and David Culler, "Telos: enabling ultra-low power wireless research," in *Proceedings of the 4th international symposium on Information processing in sensor networks*, Piscataway, NJ, USA, 2005, IPSN '05, IEEE Press.

[47] Kris Lin, Jennifer Yu, Jason Hsu, Sadaf Zahedi, David Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani Srivastava, "Heliomote: Enabling long-lived sensor networks through solar energy harvesting," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 309–309, ACM.

[48] Farhan Simjee, Devyani Sharma, and Pai H. Chou, "Everlast: Long-life, supercapacitor-operated wireless sensor node," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2005, SenSys '05, pp. 315–315, ACM.

[49] Nuno Pereira, Stefano Tennina, and Eduardo Tovar, "Building a microscope for the data center," in *Wireless Algorithms, Systems, and Applications*, Xinbing Wang, Rong Zheng, Tao Jing, and Kai Xing, Eds., vol. 7405 of *Lecture Notes in Computer Science*, pp. 619–630. Springer Berlin Heidelberg, 2012.

[50] Marie Chan, Daniel Estve, Christophe Escriba, and Eric Campo, "A review of smart homespresent state and future challenges," *Computer Methods and Programs in Biomedicine*, vol. 91, no. 1, pp. 55 – 81, 2008.

[51] "Atmel corporation, atmega1281 data sheet," 2005.

[52] "MSP430 Data Sheet, Texas Instruments howpublished="`http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=140&familyId=342`," .

[53] "Chipcon inc., chipcon cc2420 data sheet," 2003.

[54] Prabal Dutta and Adam Dunkels, "Operating systems and network protocols for wireless sensor networks," *Phil Trans R Soc A*, vol. 370, pp. 68–84, 2012.

[55] Muhammad Omer Farooq and Thomas Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[56] The TinyOS 2.x Working Group, "Tinyos 2.0," in *the 3rd international conference on Embedded networked sensor systems*, San Diego, California, USA, 2005, SenSys '05, pp. 320–320, ACM.

[57] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *the 29th IEEE International Conference on Local Computer Networks*, Washington, DC, USA, 2004, LCN '04, pp. 455–462, IEEE Computer Society.

[58] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," *Mob. Netw. Appl.*, vol. 10, no. 4, pp. 563–579, Aug. 2005.

[59] Qing Cao, T. Abdelzaher, J. Stankovic, and Tian He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, april 2008, pp. 233 –244.

[60] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan, "Tosthreads: Thread-safe and non-invasive preemption in tinyos," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2009, SenSys '09, pp. 127–140, ACM.

[61] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, New York, NY, USA, 2003, PLDI '03, pp. 1–11, ACM.

[62] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa, "Resource kernels: a resource-centric approach to real-time and multimedia systems," 1997, vol. 3310, pp. 150–164.

[63] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," *38th Annual IEEE Conference on Local Computer Networks*, vol. 0, pp. 641–648, 2006.

[64] Luca Mottola and Gian Pietro Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *Lecture Notes in Computer Science : Distributed Computing in Sensor Systems*. vol. 4026/2006, pp. 150–168, Springer Berlin / Heidelberg.

[65] Myra Dideles, "Bluetooth: A technical overview," *Crossroads*, vol. 9, no. 4, pp. 11–18, June 2003.

[66] F. Cal ì, M. Conti, and E. Gregori, "Ieee 802.11 wireless lan: Capacity analysis and protocol enhancement," 1998.

[67] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Comput. Netw.*, vol. 38, no. 4, pp. 393–422, Mar. 2002.

[68] H. Zimmermann, "Innovations in internetworking," chapter OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, pp. 2–9. Artech House, Inc., Norwood, MA, USA, 1988.

[69] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong, "Splash: Fast data dissemination with constructive interference in wireless sensor networks," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, 2013, pp. 269–282, USENIX.

[70] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th*

Annual International Conference on Mobile Computing and Networking, New York, NY, USA, 2000, MobiCom '00, pp. 56–67, ACM.

[71] Adrian Perrig, John Stankovic, and David Wagner, "Security in wireless sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 53–57, June 2004.

[72] M. Rudafshani and S. Datta, "Localization in wireless sensor networks," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, April 2007, pp. 51–60.

[73] Anthony Rowe, Vikram Gupta, and Ragunathan (Raj) Rajkumar, "Low-power clock synchronization using electromagnetic energy radiating from ac power lines," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Berkeley, California, 2009, SenSys '09, pp. 211–224, ACM.

[74] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani, "Clock synchronization for wireless sensor networks: a survey," *Ad Hoc Networks*, vol. 3, no. 3, pp. 281 – 323, 2005.

[75] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, C. Mitter and B. Heile, "IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks," 2001.

[76] Shahin Farahani, *ZigBee Wireless Networks and Transceivers*, Newnes, Newton, MA, USA, 2008.

[77] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling, "Capture effect based communication primitives: Closing the loop in wireless cyber-physical systems," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, New York, NY, USA, 2012, SenSys '12, pp. 341–342, ACM.

[78] J. Polastre, J. Hill and D. Culler, "Versatile low power media access for wireless sensor networks," *SenSys*, November 2005.

[79] Wei Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 3, pp. 493–506, June 2004.

[80] A El-Hoiydi and J-D Decotignie, "Wisemac: an ultra low power mac protocol for the downlink of infrastructure wireless sensor networks," in *Computers and Communications, 2004. Proceedings. ISCC 2004. Ninth International Symposium on*, June 2004, vol. 1, pp. 244–251 Vol.1.

[81] Venkatesh Rajendran, Katia Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient collision-free medium access control for wireless sensor networks," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2003, SenSys '03, pp. 181–192, ACM.

[82] Rowe A., Mangharam R., and Rajkumar R., "RT-Link: A Time-Synchronized Link Protocol for Energy-Constrained Multi-hop Wireless Networks," *SECON*, 2006.

[83] J.N. Al-Karaki and AE. Kamal, "Routing techniques in wireless sensor networks: a survey," *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 6–28, Dec 2004.

[84] C.E. Perkins and E.M. Royer, "Ad-hoc on-demand distance vector routing," in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, 1999, pp. 90–100.

[85] Philip Levis, Neil Patel, David Culler, and Scott Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, Berkeley, CA, USA, 2004, NSDI'04, pp. 2–2, USENIX Association.

[86] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2009, SenSys '09, pp. 1–14, ACM.

[87] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein, "Kairos: a macro-programming system for wireless sensor networks," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, Brighton, United Kingdom, 2005, pp. 1–2, ACM.

[88] Carsten Bormann, Angelo P. Castellani, and Zach Shelby, "Coap: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[89] Christos Efstratiou, Ilias Leontiadis, Cecilia Mascolo, and Jon Crowcroft, "Demo abstract: A shared sensor network infrastructure," 2010.

[90] Intel, "iMote2 Datasheet: http://www.xbow.com/Products/Product_pdf_files /Wireless_pdf/Imote2_Datasheet.pdf," .

[91] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient sensor network reprogramming through compression of executable modules," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, 2008, pp. 359 –367.

[92] Jaein Jeong and D. Culler, "Incremental network programming for wireless sensors," in *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, 2004, pp. 25 – 33.

[93] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco, "Tinylime: Bridging mobile and sensor networks through middleware," *Pervasive Computing and Communications, IEEE International Conference on*, vol. 0, pp. 61–72, 2005.

[94] Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, and Carlos Ferraz, "A message-oriented middleware for sensor networks," in *Proceedings of the 2Nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, New York, NY, USA, 2004, MPAC '04, pp. 127–134, ACM.

[95] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei, "Event detection services using data service middleware in distributed sensor networks," *Telecommunication Systems*, vol. 26, pp. 351–368, 2004, 10.1023/B:TELS.0000029046.79337.8f.

[96] Object Management Group, "Corba component model 4.0 specification," Specification Version 4.0, Object Management Group, April 2006.

[97] AL. Murphy, G.P. Picco, and G. Roman, "Lime: a middleware for physical and logical mobility," in *Distributed Computing Systems, 2001. 21st International Conference on.*, Apr 2001, pp. 524–533.

[98] Yong Yao and Johannes Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.

[99] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler, "Hood: a neighborhood abstraction for sensor networks," in *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, Boston, MA, USA, 2004, pp. 99–110, ACM.

[100] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe, "Space-time scheduling of instruction-level parallelism on

a raw machine," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, San Jose, California, United States, 1998, ASPLOS-VIII, pp. 46–57, ACM.

[101] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek, "Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, Santa Clara, California, United States, 1991, ASPLOS-IV, pp. 164–175, ACM.

[102] Kevin Klues, Vlado Handziski, Chenyang Lu, Adam Wolisz, David Culler, David Gay, and Philip Levis, "Integrating concurrency control and energy management in device drivers," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, New York, NY, USA, 2007, SOSP '07, pp. 251–264, ACM.

[103] Sheldon Finkelstein, "Common expression analysis in database applications," in *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, Orlando, Florida, 1982, SIGMOD '82, pp. 235–245, ACM.

[104] Timos K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, pp. 23–52, March 1988.

[105] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin, "On-the-fly sharing for streamed aggregation," in *to appear: Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*.

[106] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele, "Low-power wireless bus," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, New York, NY, USA, 2012, SenSys '12, pp. 1–14, ACM.

[107] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh, "Efficient Network Flooding and Time Synchronization with Glossy," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Chicago, IL, USA, April 2011, pp. 73–84, Best paper award.

[108] Wei Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2002, vol. 3, pp. 1567–1576 vol.3.

[109] Tijs van Dam and Koen Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, 2003, SenSys '03, pp. 171–180, ACM.

[110] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, New York, NY, USA, 2006, SenSys '06, pp. 307–320, ACM.

[111] M.T. Hansen, R. Jurdak, and B. Kusy, "Unified broadcast in sensor networks," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, 2011, pp. 306–317.

[112] M. Maroti and B. Kusy and G. Simon and A. Ledeczi, "The flooding time synchronization protocol," *Proc. ACM Sensys*, 2004.

[113] Ted Herman and Sébastien Tixeuil, "A distributed tdma slot assignment algorithm for wireless sensor networks," *CoRR*, vol. cs.DC/0405042, 2004.

[114] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F. Harris, and M. Zorzi, "Synapse: A network reprogramming protocol for wireless sensor networks using fountain codes," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, 2008, pp. 188 –196.

[115] Nithya Ramanathan, Thomas Harmon, Laura Balzano, Deborah Estrin, Mark Hansen, Jenny Jay, William Kaiser, and Gaurav Sukhatme, "Designing wireless sensor networks as a shared resource for sustainable development," in *in Information and Communication Technologies and Development*, 2006.

[116] Yang Yu, Bhaskar Krishnamachari, and V.K. Prasanna, "Issues in designing middleware for wireless sensor networks," *Network, IEEE*, vol. 18, no. 1, pp. 15 – 21, 2004.

[117] "http://www.nanork.org/wiki/nanocf," .

[118] D. B. Johnson, D. A. Maltz, and J. Broch, "DSR: the dynamic source routing protocol for multi-hop wireless ad hoc networks," *Ad hoc networking*, vol. 5, pp. 139172, 2001.

[119] Matt Welsh and Geoff Mainland, "Programming sensor networks using abstract regions," in *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, San Francisco, California, 2004, pp. 3–3, USENIX Association.

[120] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, Boston, Massachusetts, 2002, pp. 131–146, ACM.

[121] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI*, p. 13, 2004.

[122] Christine Jardak, Janne Riihijärvi, Frank Oldewurtel, and Petri Mähönen, "Parallel processing of data from very large-scale wireless sensor networks," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, Illinois, 2010, HPDC '10, pp. 787–794, ACM.

[123] O. Olguín et al., "Sociometric badges: Wearable technology for measuring human behavior," 2007.

[124] A. Kandhalu, K. Lakshmanan, and R.R. Rajkumar, "U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 350–361.

[125] Ken Kennedy and John R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[126] Prabal Dutta, David Culler, and Scott Shenker, "Procrastination might lead to a longer and more useful life," 2007.

[127] L. Comtet, *Advanced Combinatorics*, Reidel, Dordrecht, 1974.

[128] CPLEX, "Ibm ilog cplex optimizer," `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`, 2013.

[129] Gurobi, "Gurobi optimizer," http://www.gurobi.com/products/gurobi-optimizer/gurobi-overview, 2013.

[130] Z. Deng and J.W.-S. Liu, "Scheduling real-time applications in an open environment," *1997 IEEE 18th Real-Time Systems Symposium*, vol. 0, pp. 308, 1997.

[131] Giuseppe Lipari and Enrico Bini, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, pp. 257–269, April 2005.

[132] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, july 2008, pp. 181 –190.

[133] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th Design Automation Conference*, New York, NY, USA, 2010, DAC '10, pp. 731–736, ACM.

[134] NHS, "http://users.ece.cmu.edu/~vikramg/nhs.htm," 2013.

[135] S. Ramanathan, "A unified framework and algorithm for channel assignment in wireless networks," *Wirel. Netw.*, vol. 5, no. 2, pp. 81–94, Mar. 1999.

[136] Jianping Pan, Y. Thomas Hou, Lin Cai, Yi Shi, and Sherman X. Shen, "Topology control for wireless sensor networks," in *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, New York, NY, USA, 2003, MobiCom '03, pp. 286–299, ACM.

[137] A. Yao, "On constructing minimum spanning trees in k-dimensional spaces and related problems," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736, 1982.

[138] David R. Karger, Philip N. Klein, and Robert E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *J. ACM*, vol. 42, no. 2, pp. 321–328, Mar. 1995.

[139] Roger Wattenhofer and Aaron Zollinger, "Xtc: A practical topology control algorithm for ad-hoc networks," in *In 4th International Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN*, 2003.

[140] "http://www.nest.com (viewed Aug 20, 2014)," .

[141] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl, "The home needs an operating system (and an app store)," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, USA, 2010, Hotnets, pp. 18:1–18:6, ACM.

[142] Vikram Gupta, Eduardo Tovar, Jose Marinho, and Ragunathan(Raj) Rajkumar, "Compiler-assisted strategic redundancy elimination across multiple applications for sensor networks," *Under submission to the International Journal of Embedded Systems, Inderscience Publishers*, 2014.

[143] Andrew S. Tanenbaum and Robbert Van Renesse, "Distributed operating systems," *ACM Comput. Surv.*, vol. 17, pp. 419–470, December 1985.

[144] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava, "A dynamic operating system for sensor nodes," in *the 3rd international conference on Mobile systems, applications, and services*, Seattle, USA, 2005, MobiSys '05, pp. 163–176, ACM.