REMOTE REPROGRAMMING OF WIRELESS SENSOR NETWORKS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rajesh K. Panta

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2010

Purdue University

West Lafayette, Indiana

Dedicated to my parents for their love, support and encouragement.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Saurabh Bagchi for his guidance, support, advice and insightful inputs throughout my research and academic works. His constant motivation and encouragement helped tremendously in this research effort. His enthusiasm, dedication and patience are truly admirable. I am also thankful to Prof. James V. Krogmeier, Prof. Samuel P. Midkiff and Prof. Vijay Raghunathan for serving in my Doctoral committee. I would also like to thank Prof. Samuel P. Midkiff for his inputs on the incremental reprogramming part of this thesis. I am also thankful to Prof. Vijay Raghunathan for his advice regarding Sensor Operating System (SOS). I am grateful to Dr. Issa Khalil, Mark D. Krasniewski and John Mastarone for their help during the early stages of this work.

My special thanks go to Bhumika for her unconditional love and patience, Aditi and Rewa for making life so beautiful, and Raju and Barsha for their help and support. And most of all, my parents, Babu Krishna Panta and Bhagawati Panta, who always emphasized education, encouraged me to aim for higher goals, and guided me to become a better person. I would not be where I am today without their love, support and sacrifice.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Panta, Rajesh K. Ph.D., Purdue University, May 2010. Remote Reprogramming of Wireless Sensor Networks. Major Professor: Saurabh Bagchi.

In recent years, advances in hardware and software tools have led to many real-world sensor network deployments. Management of already deployed sensor networks is a very important issue. One of the crucial management tasks is that of software reconfiguration. During the lifetime of a sensor network, software running on the sensor nodes may need to be changed for various reasons like correcting software bugs, modifying the application to meet the changing environmental conditions in which the network is deployed, adapting to evloving user requirements, etc. The frequency of software updates is very high in sensor networks due to various reasons — harsh and unpredictable environments in which the sensor nodes are deployed, time-varying nature of the wireless channel, lack of robust tools for developing software, interference in the unlicensed ISM band, topology change casued by node mobility, battery outages, etc. Since a sensor network may consist of hundreds or even thousands of nodes which may be situated at places which are difficult or, sometimes, impossible to access physically, remote reprogramming of sensor networks is essential. This thesis presents energy efficient and fast reprogramming services for wireless sensor networks. Sensor networks are often battery-powered and need to be operated unattended for long periods of time. Radio transmission is often the most energy-expensive operation in sensor networks. Since energy is a very scarce resource, this thesis focuses on conserving energy during reprogramming. Also, since the performance of the network may be degraded, or even reduced to zero, during software update process, the reprogramming techniques proposed here minimize reprogramming time significantly compared to existing reprogramming protocols.

1. INTRODUCTION

Wireless sensor networks represent a new class of networked computing devices. A typical wireless sensor network consists of a large number of small, low-power, low-cost, battery-powered wireless sensors and (optionally) actuators that integrate computation, communication, sensing, and actuation together. Due to the small size of these devices, they can be deployed unobtrusively in a wide range of environments, enabling data collection at a fidelity and scale that was previously unthinkable. Using wireless communication, these devices coordinate among each other to perform sensing tasks typically over large temporal and spatial scales. Additionally these devices may also be equipped with actuation capabilities, which allow sensor networks not only to sense the physical phenomenon, but also power the actuators to affect the sensed environment. This forms a distributed feedback loop that has the potential for efficiently controlling geographically distributed processes at a scale that was until recently infeasible.

The proliferation of many small sensor hardware [1–9] and software [10–15] platforms have opened up possibilities to embed sensors in large and uncontrolled environments for a wide range of disciplines. In recent years, many sensor network deployments have been reported in the literature—environmental monitoring [16–25], structural monitoring [26–28], habitat monitoring [29–32], home and building automation [33, 34], vehicular networks [35–37], data center monitoring and control [38], disaster management [39], shooter localization [40], intrusion detection [41, 42], remote medicine and patient monitoring [43–45], etc. Management of already deployed sensor networks has thus become very important. One of the crucial management issues is that of software reconfiguration. Once the sensor network is deployed, it may be necessary to change the software running on the sensor nodes for various reasons—to fix software bugs, to add new features to sensor network application, to fine-tune sens-

ing algorithms, etc. Before installing software on the sensor nodes, the program code is first developed and compiled on a resource-rich host computer because the sensor nodes are resource-constrained (in computation, memory, and energy) devices. Traditionally, the sensor network is reprogrammed by bringing the sensor nodes from the deployment field to the laboratory, connecting them one by one to the host computer, and uploading the code to each node via serial or parallel port (In System Programming or ISP). This method is obviously tedious and expensive—both in terms of time and money. Furthermore, ISP may be very difficult or sometimes impossible since the sensor nodes may be situated at places which are difficult or impossible to access physically (e.g. deep ocean, enemy territory, thickets [46], bird burrows [21], glaciers [47], volcanoes [48], tops of light poles [22, 49, 50], etc.). *Remote reprogramming of wireless sensor network* is thus necessary so that the new version of the software can be wirelessly transferred to all the sensor nodes in the network, without the need to be in physical contact with the nodes.

1.1 Motivation for remote reprogramming

Wireless sensor networks need software updates in a wide range of scenarios ranging from evaluation and implementation of new functionalities of the existing application to complete replacement of the existing application by a new one. This section presents some of the typical reasons for remote reprogramming of sensor networks.

1.1.1 Shortening software development and testing phase

Before a software application is deployed in a real-world sensor network, it goes through an iterative process where the software is written, tested, and debugged many times in laboratory testbeds. The ability to reprogram all the sensor nodes in a testbed network quickly thorugh wireless medium shortens the software development and evaluation phase [51].

1.1.2 Fixing software bugs

The deployed software on a network may also need to be changed to correct software bugs. In spite of rigorous testing in laboratory testbed environments, sensor network applications exhibit higher failure rate in real-world delpoyments comapared to traditional computer software. There are many reasons for this behaviour. In the programming environment for sensor nodes, non deterministic bugs, which are triggered by a difficult-to-recreate combination of environmental and protocol events, are quite likely. These are discovered during the operational phase when the sensor nodes are embedded in the environment, rather than during testing in the laboratory [52]. The harsh and unpredictable nature of the environment often exacerbates this problem. Below, we present some of the software related problems experienced by practical sensor networks.

In the LOFAR-agro deployment [53], the complex interaction among different independently developed protocols (MintRoute [54] and T-MAC [55]) prevented efficient data logging, resulting in data yield of only 2%. In a surveillance application [42], some extreme environmental conditions caused software faults on some nodes triggering false events and resulting in fast draining of the batteries. Similarly, in a vineyard monitoring application [56], unknown software problems caused severe packet losses. Their experiments in lab environment had a data yield of 90%, while the actual deployment resulted in a data yield of only 77%, even with 5x redundancy [52]. The Great Duck Island deployement [32] also faced significant packet loss, partly due to inability of software to adjust to clock drift among nodes and unexpected humidity sensor readings. Many other sensor network deployments [57–61] have reported failure or poor performances due to problems associated with the software running on the sensor nodes.

Although many tools and approaches [52, 62–75] have been proposed to build robust software for sensor nodes, they are far from achieving their goals. This is a very challenging problem because of various reasons. First, limited memory means

that sensor nodes cannot store enough system logs and checkpoints. Second, due to the distributed nature of the sensor network algorithms, sensor nodes need to communicate with each other, and possibly with the base station, to pinpoint the cause of the problem. Third, sensor nodes have limited energy supply and hence to increase its lifetime, the number of radio communications need to be minimized.

In short, compared to traditional computer networks, the software for the sensor nodes are highly failure prone. Thus, software updates are necessary to fix these problems.

1.1.3 Adapting to network failures

Extreme and unexpected environmental conditions can also trigger hardware and network failures. Network link qualities are extremely sensititive to many environmental factors. Since most of the sensor networks operate at unlicensed ISM band, they are also vulnerable to interference from other nearby transmissions. For example, the 2.5 GHz bandwidth used by IEEE 802.15.4 [76] based low power radios used by many sensor nodes is also shared by other wireless systems like IEEE 802.11 (Wi-Fi [77]), IEEE 802.15.1 (Bluetooth [78]), and even by microwaves. These cause failures in wireless communications and in some cases may cause more serious problems like network partitioning, inability to respond to critical events, etc. As a result, the software running on these nodes need to be modified to make them tolerate such failures.

A recent deplyoment of sensor network in Swiss Alps showed that wireless communication worked well during the day and the night but failed during mornings and evenings [79]. Due to different response characteristics of the processor and radio oscillators to varying temperature, the unpredictable temperature fluctuations caused the clocks to drift too much [79, 80]. In CSOnet sensor network deployment [81], the reliability of the wireless links is observed to decrease due to increasing foliage

in spring and summer, and the communication protocol has to be changed subtly to achieve the same end-to-end reliability for the data.

1.1.4 Fine-tuning the application

Large scale sensor networks may be deployed for long periods of time during which the requirements from the network or the environment in which the nodes are deployed may change. Also, the requirement of the user may change over time. The change may necessitate uploading a new code or retasking the existing code with different sets of parameters. In many cases, an application developer has a limited or no familiarity of the environment in which the sensor nodes will be deployed, eg. deep ocean, enemy territory, or wild nature. In these environemnts, software development may follow an ad-hoc approach—deploy software based on predicted nature of the environment, use collected data and the observed performance of the software as a feedback to develop more effective algorithms, and so on [82]. Remote wireless rerprogramming is crucial in these scenarios.

1.1.5 Complete application replacement

There are many situations where the application running on the sensor nodes may need to be replaced with a completely new one. In [83], a sensor network is used to detect forest fire. When a fire event is detected, the fire fighters need to run a search and rescue application as well as a fire tracking application [51]. Preloading all possible applications in memory-constrained sensor nodes may not be scalable. Hence, reprogramming the sensor network with the new application can be more economical and scalable. A sensor network may be deployed to monitor vibrations of the building structure due to, say, earthquake. Later on, instead of using a new network, the user may want to use the same network for implementing smart lighting system. Software updates are essential in these scenarios.

Because of the above mentioned reasons, practical sensor networks need to be reprogrammed more frequently than the traditional computer networks. The exact reprogramming frequency depends on the nature of the deployment. For example, the Kansei testbed [84] is reprogrammed approximately every 3 months. Also, this testbed undergoes major updates with significant new features added approximately every 6 months. Each major update is generally followed by a burst of other small updates in a short period of time, to fix the bugs associated with the major update. Kansei network can also be used by researchers and developers from all over the world to conduct experiments with their applications. Every time a user submits a new application, it needs to be disseminated to (a subset of) nodes in the network. The Kansei testbed was reprogrammed 5251 times with applications submitted by the users between October 2005 to May 2008. In James Reserve deployment [85], sensor nodes needed several software updates every day in field-testing phase. After that, one or two updates were needed every week. Finally, as the software got more stable, updates were required approximately every 4 to 6 weeks.

1.2 Requirements of remote reprogramming

Remote reprogramming in sensor networks poses several challenges. A primary requirement is that the reprogramming be done while the nodes are *in situ*, embedded in their sensing environment. To ensure correctness of the system, it is essential that the code update be 100% reliable and reach all the nodes that it is destined for. Each node being reprogrammed must receive each and every bit of the code correctly. Compared to traditional computer networks, sensor networks are more failure prone. The network topology cannot be assumed to be static. It can change frequently in many sensor network deployments because of node mobility, time-varying wireless channel conditions, physical environmental changes, node failures, battery outages, unpredictable harsh environmental conditions, etc. A program image is relatively

large for the low-bandwidth wireless radio. Thus it is very challenging to ensure that all nodes are updated with the latest version of the code all the time.

Two important performance metrics in reprogramming are time and energy. The code upload should be fast since the networks functionality is likely degraded, if not reduced to zero, during the reprogramming period. Sensor networks often need to be operated unattended for long periods of time. Once the network is deployed, it may be very difficult or sometimes impossible to replace the batteries. Thus, reprogramming should incur as little energy as possible. Often radio transmission is the most energy expensive operation in sensor networks. Research studies have shown that transmitting a single bit consumes the same energy as executing 1000 instructions [82,86,87]. Code delivery has to be done efficiently to minimize redundant transmissions due to multiple senders and extra retransmissions due to link losses or collisions. Also, sensor nodes have limited energy supply, bandwidth, and memory. So it is important to minimize these resources consumed for network reprogramming.

It is conceivable that the process of code upload will be infrequent for many deployments and therefore it may appear that its resource consumption need not be optimized. However, consider that the sensor network environment has inherent unreliability in the wireless links that may have transient failures. Thus the environment is dynamic with nodes coming in and out of periods of disconnectedness. Also, the network may have nodes added after the initial deployment while new code may be injected at arbitrary points in time. Since in most deployments, the sensor network is expected to operate over extended periods of time, it is possible that the parameters for the application, such as the monitoring period, change. All of these necessitate retasking some or all of the sensor nodes. The code dissemination therefore cannot be considered a one shot process and it becomes important to minimize the resource consumption used in network reprogramming.

1.3 Thesis outline

In Chapter 2, we present some related work. The next few chapters present novel middleware services for reprogramming sensor networks—Stream [88, 89] (Chapter 3), DStream [50] (Chapter 4), Zephyr [90] (Chapter 5), and Hermes [91] (Chapter 6). Chapter 7 presents a protocol called Varuna that minimizes the energy consumption during *quiescent state*—time period during which no code upload is actually being done.

Stream: The amount of information that needs to be transferred wirelessly during reprogramming greatly affects the reprogramming time as well as energy. Stream significantly reduces the time and energy required to reprogram the network by minimizing the amount of information that needs to be transferred wirelessly for reprogramming the network. Deluge [92] is the standard reprogramming protocol for TinyOS [14] based applications. Deluge attaches the user application with the entire Deluge reprogramming protocol and sends them as one big image. This is to make sure that Deluge is always running on the sensor nodes so that they are receptive to future code updates. Note that sensor nodes are generally not capable of multitasking as it is very expensive because of severe resource-constraints. Stream attaches the user application with a small component instead of the entire reprogramming protocol. This component is capable of listening to future code update messages and running the reprogramming protocol on the nodes on an as-needed basis.

DStream: DStream, an extension to Stream protocol, is capable of using either single hop or multi hop modes of reprogramming based on current network conditions (like link reliabilities among the nodes). In spite of the current research trend towards multi hop reprogramming, we show, through mathematical analysis, real testbed experiments and simulations, that under certain network conditions, single-hop reprogramming can be faster and more energy efficient than multi hop mode. DStream is equipped with both the single and multi hop modes of reprogramming so that a choice can be made between the two based on the current network conditions.

Zephyr: Zephyr is an incremental reprogramming protocol that exploits the fact that in real world scenario, the software running on the sensor nodes evolves with incremental changes to the functionality. Zephyr significantly reduces reprogramming time and energy by wirelessly transferring only the difference between the old and new versions of the software, rather than the entire new software. The sensor nodes build the new image using the difference and the old image. Zephyr uses modified Rsync [93] algorithm to compare the two binary images at the byte level and find the difference between them. We show that the comparison at the byte level alone is not sufficient to produce a small difference. Modifications at the application level are also necessary. Zephyr uses function call indirections to mitigate the effects of function shifts. This increases the similarity between the old and new versions of the software. As a result, the difference between them is small, and reprogramming time and energy are significantly reduced.

Hermes: One of the major problems of an incremental reprogramming scheme is that software modification can cause the global variables to be shifted. As a result, the difference between the old and the new verions of the software becomes large. Hermes eliminates the global variable shift problem in incremental reprogramming systems. Furthermore, in Zephyr, the function call indirections introduce software latency. Heremes eliminates this problem and still avoids the problem due to function shifts.

Varuna: Information dissemination protocols used in wireless ad-hoc and sensor networks incur energy expenditure not only during the data-item dissemination phase, but also during the steady-state when no dissemination is actually being done. The need for energy expenditure in the steady state arises from transient wireless link failures, incremental node deployment, and node mobility. To ensure that all nodes are up-to-date all the time, existing dissemination protocols cause sensor nodes to periodically advertise their metadata (e.g. the version number of the data-item that the node currently has) in the steady state. Thus, the steady state energy cost increases linearly with the steady state time duration, the most dominant phase in a node's

lifetime. Chapter 7 shows that the steady state energy cost can quickly outweigh the actual dissemination cost in practical systems. Varuna is a maintenance algorithm that incurs fixed maintenance cost, independent of the steady state duration.

2. RELATED RESEARCH

The question of reconfigurability of sensor networks has been an important theme in the research community. Program execution models affect software reconfigurability. In recent years, various programming environments have been proposed for sensor networks. We discern three streams of work in this regard. First, is the class of work that provides virtual machine abstractions on sensor nodes. Second, is the design for reconfigurability in sensor operating systems that do not support dynamic linking and loading. Third, is reconfigurability in systems that do support dynamic linking and loading. We discuss these three streams in order here. Many scripting language based systems have been proposed for embedded systems [51], including BASIC variants, Python interpreters [94], and TCL machines [95]. They are intended for platforms with more resources than typical sensor networks. We do not include them in our discussion.

2.1 Virtual machines

A common method used for reducing the cost of transmitting program code is a virtual machine. Several systems, such as Mate [96], VM* [97], ASVM [98] and [99] provide virtual machines that run on resource-constrained sensor nodes. They enable efficient code updates, since the code that runs on these virtual machines is more compact than the native code. However, they trade off, to different degrees, less flexibility in the kinds of tasks that can be accomplished through virtual machine programs and less efficient execution than native code.

2.2 Native code with no support for loadable modules

Most of the sensor network systems use native code that is executed directly by the microcontroller. TinyOS is the primary example of an operating system that does not support loadable program modules. There are several protocols that provide reprogramming with full binaries, such as XNP [100], MNP [101], Deluge [92], Freshet [102], etc. These protocols fall under full image replacement category because the entire program image is transmitted to the sensor node. These protocols focus on reliably and efficiently distributing the program image using the unreliable radio links.

Reliable multicast in unreliable environments, such as ad-hoc networks, can be achieved by epidemic multicast protocols based on each node gossiping the message it received to a subset of neighbors [103]. This class of protocols gives probabilistic guarantee for the update to reach all the group members. The probability is monotonically increasing with the fanout of each node (the number of neighbors to gossip to) and the quiescence threshold (the time after which a node will stop gossiping to its neighbors). By increasing the quiescence threshold, the reliability can be made to approach 1, which is the basic premise behind all the epidemic based code update protocols in sensor networks — Deluge, MNP, and Freshet.

The push-pull method for data dissemination through the three way handshake of advertisement-request-code has been used previously in sensor networks with sensed data taking the place of code. Protocols such as SPIN [104] and SPMS [105] rely on the advertisement and the request packets being much smaller than the data packets and the redundancy in the network deployments which make several nodes disinterested in any given advertisement. However, in the data dissemination protocols, there is only suppression of the requests and the data sizes are much smaller than the entire binary code images.

The earliest network reprogramming protocol XNP [100] only operated over a single hop and did not provide incremental updates of the code image. The Multihop

Over the Air Programming (MOAP) protocol extended this to operate over multiple hops [106]. MOAP introduced several concepts which are used by later protocols, namely, local recovery using unicast NACKs and broadcast of the code, and sliding window based protocol for receiving parts of the code image. However, MOAP did not leverage the pipelining effect with segments of the code image. That is a node has to download the program image completely before sending it to other nodes in the network.

The three protocols that are substantially more sophisticated than the rest and define the state-of-the-art today are Deluge [92], MNP [101], and Freshet [102]. All use the three way handshake for locally propagating the code. Deluge was the earliest and laid down some design principles used by the other two. It uses a monotonically increasing version number, segments the binary code image into equal sized pages, and pipelines the different pages across the network. It builds on top of Trickle [107], a protocol for a node to determine when to propagate code in a one hop case. The code distribution functions through a three-way handshake protocol of advertisement, request, and broadcast code. The operation of each node is periodic according to a fixed size time window. The first part of the window is for listening to advertisements and requests and sending advertisements. The second part of the window is for transmitting or receiving code corresponding to the received requests. Within the first part of the time window, a node randomly selects a time at which to send an advertisement with metadata containing the version number, the number of complete pages it has, and the total number of pages in the image of this version. When the time to transmit the advertisement comes, the node sees whether it has heard a threshold number of advertisements with identical metadata, and if so, it suppresses the advertisement. When a node hears code that is newer than its own, it sends a request for that code and the lowest number page it needs, to the node that advertised the new code. In the second part of the periodic window, the node transmits packets with the code image, corresponding to the pages for which it received requests. A receiving node only fills its pages in monotonically increasing order thereby eliminating the need for

maintaining large state for missing holes in the code. For receiving the code, each node uses the shared broadcast medium that allows overhearing and can fill in a page requested by a neighbor.

The design goal of MNP [101] is to choose a local source of the code which can satisfy the maximum number of nodes. They provide energy savings by turning off the radio of non-sender nodes. Freshet [102] is different in aggressively optimizing the energy consumption for reprogramming. It introduces a new phase called blitzkrieg when the code update is started from the base node. During the blitzkrieg phase, information about the code and topology (primarily the number of hops a node is away from the wave front where the code is at) propagates through the network rapidly. Using the topology information each node estimates when the code will arrive in its vicinity and the three way handshake will be initiated — the distribution phase. Each node can go to sleep in between the blitzkrieg phase and the distribution phase thereby saving energy. Freshet also optimizes the energy consumption by exponentially reducing the metadata rate during conditions of stability in the network when no new code is being introduced, called the quiescent phase. The time required to reprogram the network can be reduced by using multiple channels for code dissemination [108]. Also, there have been few works which attempt to ensure that the code upload process is secure [109–116]. Some systems consider reprogramming in scenarios where a sensor node may be running multiple applications [117, 118].

In recent years, the research focus has shifted towards multi hop reprogramming. In Chapter 4, we discuss the scenarios in which single hop reprogramming can be faster and more energy efficient than multi hop mode and present a protocol called DStream which provides both single and multi hop modes of reprogramming. To the best of our knowledge, all of the existing reprogramming protocols provide either single or multi hop reprogramming features, but not both. Importantly existing work is silent on the choice between the two approaches for different deployment conditions. Chapter 4 also discusses how link reliabilities affect the choice between these two modes of reprogramming. There have been some studies which show how low link reliabilities

cause problems in multi-hop networks. [119] showed that shortest path algorithm in a network with lossy links selects a path with poor reliability. In [120], the authors evaluate Deluge and MNP for different densities and packet organizations. But as far as we know, there has been no prior work to study the effect of parameters like link reliabilities on the performance of multi-hop reprogramming. In Chapter 4, we show how poor link qualities adversely affect multi-hop reprogramming making the alternate single hop reprogramming approach attractive.

2.3 Incremental reprogramming systems

The reprogramming protocols discussed above transmit the complete code image during reprogramming. In addition to these full image replacement algorithms, there are some techniques proposed in the literature which reduce the number of packets transmitted during reprogramming by comparing the new code with the previously installed software and transmitting only the difference. Using a diff-like approach, Reijers and Langendoen [87] compute the diff script which captures the difference between the old and the new code and consists of *copy, insert, address repair and address patch operations*. This approach reduces the network traffic but the drawback of their approach is that the address patching is dependent on the regular structure of the instruction set architecture and the authors demonstrate the protocol for a specific customized node called EYES [4]. Also, their work is focused on the encoding scheme and does not demonstrate a fully functional implementation of reprogramming. Jeong and Culler [121] describe an incremental network reprogramming protocol which uses the Rsync algorithm [93] to find the blocks of the code that are identical in the new and the old code images. A drawback of their approach is that their protocol is meant for single hop reprogramming. Also, since they do not use the knowledge about the application structure, even small code shifts can result in a large number of address patches leading to high bandwidth consumption. Koshy and Pandey [122] have the design goal of keeping address patches to a small number. They use slop regions after

each function in the application so that the function can grow. However, the slop regions lead to fragmentation and inefficient usage of the Flash and the approach only handles growth of functions up to the slop region boundary. The authors in [123] present a mechanism for linking components on the sensor node without support from the underlying OS. This is achieved by sending the compiled image of only the changed component along with the new symbol and relocation tables to the nodes, where the new image is created. This has been demonstrated only in an emulator [124] and makes extensive use of Flash. Also, the symbol and relocation tables can grow very large resulting in large updates. Some schemes [82, 125] change the compiler to increase the similarity of two versions of the software.

Reconfigurability is simplified in operating systems that support linkable modules. The prominent examples are SOS [10], Contiki [15], and RETOS [11]. In all of these, individual modules can be loaded. Specific challenges exist in the matter of reconfiguration in individual protocols. SOS uses position independent code and due to architectural limitations on common embedded platforms, the relative jumps can be within a certain offset only (such as 4 KB for the Atmel AVR platform). Contiki is an operating system developed for resource constrained systems like sensor networks. Based on Contiki, Dunkels et al. [51] describe a reprogramming approach using dynamic linking on the sensor nodes. Contiki disseminates the symbol and relocation tables, which may be quite large for cases where many functions are referenced in the application. Unlike these incremental approaches, we propose Stream (Chapter 3) that uses the full image replacement technique and still manages to greatly reduce the amount of data transmitted during reprogramming. We also propose incremental reprogramming protocols Zephyr (Chapter 5) and Hermes (Chapter 6) that do not require dynamic linking feature from the underlying operating system and the transfer of symbol and relocation tables.

3. STREAM: LOW OVERHEAD REPROGRAMMING PROTOCOL

In recent years, some protocols have been proposed for reprogramming in sensor networks, the state-of-the-art being defined by three protocols—Deluge [92], MNP [101], and Freshet [102]. Common to the three protocols is the notion of transferring the code image in chunks of pages on a hop-by-hop basis with each node disseminating code to its immediate neighbor through a three-way handshake of advertisement, request, and actual code transfer. MNP and Freshet build on Deluge and respectively optimize the transfer through judicious sender selection for dense networks and sleep-awake protocols for large networks.

The critical problem that besets all three protocols is what is transferred. Common intuition would be to transfer just what is needed, in other words, the application image (or the image of the updates to the application). However, each protocol transfers the image of the entire reprogramming protocol together with the minimally necessary part. The reason behind this is the fact that the reprogramming component should be running on the sensor nodes all the time for them to be receptive to future code updates. However, the existing sensor network operating systems do not support multitasking feature and, as a result, the reprogramming component should be attached to the application image. Since the reprogramming protocols are of considerable complexity, the inflation in the program image size that gets transferred over the wireless medium increases greatly. The exact amount of increase is application specific — for a simple stand-alone application (without wireless communication feature) of 1 page, the increase is 20 folds, while for a communicating application of the same size, the increase is 11 folds. In a sensor network environment, this is problematic. First, the network links are prone to transient failures and yet, the code upload process needs to be 100% reliable. Second, the networks are envisaged to be large and the cost

of larger image is incurred at every hop and does not get amortized. Third, it puts pressure on multiple scarce resources of a node – communication bandwidth leading to communication contention, and program Flash memory. The authors of Deluge argue convincingly that it is difficult to improve over Deluge the rate of transfer over the wireless link. Therefore, the logical approach appears to be to optimize what needs to be transferred, keeping the basic mode of transfer the same as in Deluge.

This thinking gives rise to our protocol called Stream which transfers close to the minimally required image size by segmenting the program image into an application image and the reprogramming protocol image. It transfers over the wireless link the former with a minimal addition. It pre-installs in each node, before deployment, the reprogramming protocol image. Stream utilizes the ability to segment the external Flash memory into multiple images and stores the two in two different image areas. An application is modified by linking it to a small component called *Stream-ApplicationSupport (Stream-AS)* while *Stream-ReprogrammingSupport (Stream-RS)* is pre-installed in each node. Stream-AS is generic and can be inserted in any TinyOS application through the insertion of two lines of nesC [126] code. Stream-RS builds on Deluge to operate in the changed mode. Overall, Stream's design principle is to limit the size of Stream-AS and providing it the facility to switch to Stream-RS when triggered by a code update related message. The advantage afforded by Stream is demonstrated over Deluge, though it can apply to any of the three protocols, since the problem Stream addresses is shared by each. What would change in applying to a different protocol is that Stream-RS will be based on that protocol.

There are several challenges to implementing the basic idea of Stream in the mote platform. First, the node that has been updated with the recent code needs to remain receptive to future code updates. Thus, it cannot be running just the application. The mote platform does not support multi-tasking and therefore the two programs (reprogramming protocol and application) cannot be executing concurrently. A design option we explored was to pre-install the reprogramming protocol components in the node and dynamically link it to the application to create a single executable

image once the application is uploaded. However, TinyOS [14] does not provide a linking facility on the node itself. As we will show in chapter 5, although some operating systems like SOS [10] and Contiki [15] support dynamic linking on the nodes, some sensor node architectural issues make them impractical or very energy-intensive. Second, it is unreasonable to assume that the code update will always occur according to a preset schedule in which case the node could have queried the base station for it. Third, Stream has to consider the possibility that new nodes may be introduced into the network and may query a given node for coming up-to-date with the latest version of the code. Thus a node cannot be content to handle just its own need for staying up-to-date.

When a node has received all its code update, Stream optimizes the steady-state energy expenditure by switching from a push-based mechanism (where the node periodically sends advertisements) to a pull-based mechanism where a newly inserted node requests for the code. The benefit of Stream shows up in fewer number of bytes transferred over the wireless medium leading to increased energy savings and reduced delay for reprogramming. To further reduce the energy used in reprogramming the sensor network, Stream causes the nodes to be involved in reprogramming only when they are actually required to do so, i.e., a node is neither woken up nor switched over from its application duties till the new code has reached the neighborhood and the node has to be involved in the three way handshake for getting the code. Freshet [102] reduces the reprogramming energy by cleverly estimating how long a node can sleep before the new code, after being injected at one point in the network, arrives its vicinity. However, due to the variability of the wireless channel, the estimate made by Freshet based on the hop count is often inaccurate. An inaccurate estimate either causes higher energy expenditure (if the time estimate is too low) or higher delay in completing the reprogramming (if the estimate is too high). Stream achieves the goal without needing to estimate the time, but by rebooting the node from Stream-RS for the purpose of reprogramming only when the new code arrives at one of its neighbors. As a result, the user application running on the node can put the node to sleep till

the time to reboot comes. This opportunistic sleeping feature of Stream is useful in conserving the energy in resource constrained sensor networks, especially for large networks where the amount of time to disseminate the code can be quite significant (tens of minutes). Coupled with this fact is the observation that reprogramming is not a one-time task, but rather is done periodically, and quite frequently, in some networks.

We demonstrate the above mentioned claims by implementing Stream in nesC [126] for the Mica2 [1] mote platform. We conduct experiments with Deluge and Stream on a real small-sized testbed (of up to 16 nodes) in linear and grid topologies. The output metrics we measure are number of bytes transferred (which relates to the energy spent) and the delay. We see that Deluge requires 63% to 98% more reprogramming time and transfers 75% to 132% more number of bytes than Stream for the grid topologies. To evaluate Stream for larger sized networks, we use the TOSSIM [127] simulation environment. We present a mathematical analysis to evaluate the performance of Stream and compare it to the ideal case when exactly the application image is transferred.

3.1 Stream Design

3.1.1 Design Approach

Stream builds on the code distribution method of Deluge. It optimizes the number of bytes that needs to be disseminated over the wireless medium so that instead of transferring the entire Deluge component along with the new application, only a small subset of reprogramming functionality is included in the program image. The idea is to have all nodes in the network be pre-installed with the *Stream-ReprogrammingSupport (Stream-RS)* component that includes the complete functionality for network reprogramming. Stream-RS is installed as image 0 in external flash memory. The application image augmented with the *Stream-ApplicationSupport* (*Stream-AS*) component that provides minimal support for network reprogramming

is installed as image 1 in external flash memory. The addition to the size of the program image over the application image size with Stream is significantly less than in the Deluge case. When a new program image is to be injected into the network, all the nodes in the network running image 1 reboot from image 0 and the new image is injected into the network using Stream-RS. The new image again includes Stream-AS and we avoid the entire Deluge component from being transferred to all the nodes each time the network needs to be reprogrammed. This modification does not entail modification of the application on the part of the user. Instead of adding the Deluge component, she adds the much smaller component (Stream-AS) to her application. Both are localized two line changes in the application code.

The saving in terms of the number of pages transferred is quite significant. The exact figure depends on the application. Any application that uses radio communication will need to add about 11 more pages if Deluge is used while Stream-AS adds only one more page. We stress that this benefit is demonstrated here for Deluge, but applies equally to all the current network reprogramming protocols since each transfers the entire protocol image along with the application image.

3.1.2 Protocol Description

Consider that initially all nodes have Stream-RS as image 0 and the application with Stream-AS as image 1 in their external flash memory. Each node is executing the image 1 code in program memory. The node that initiates the reprogramming is attached to a computer through the serial port and is called the base node.

Following is the description of how Stream works when a new user application, again with the Stream-AS component added to it, has to be injected into the network.

1. In response to the reboot command from the user, all nodes in the network reboot from image 0. This is accomplished as follows:

(a) The base node executing image 1 initiates the process by generating a command to reboot from image 0. It broadcasts the reboot command to its one hop neighbors and itself reboots from image 0.

(b) When a node running the user application receives the reboot command, it rebroadcasts the reboot command and itself reboots from image 0.

2. Once the reboot command reaches all nodes, all nodes start running Stream-RS. Then the new user application is injected into the network using Stream-RS.

3. Stream-RS starts to reprogram the entire network. It does so by using the three way handshake method (like in Deluge [92]) where each node broadcasts the advertisement about the code pages that it has. When a node hears the advertisement of newer data than it currently has, it sends a request to the node advertising newer data. Then the advertising node broadcasts the requested data. Each node maintains a set S containing the node ids of the nodes from which it has received the requests.

4. Once the node downloads the new user application completely, it performs a single hop broadcast of an ACK indicating it has completed downloading.

5. Upon receiving the ACK from a node, it removes the id of that node from the set S.

6. When the set S is empty and all the images are complete (by complete we mean that all pages of all images have been downloaded), the node reboots from image 1. So, after sometime the entire network is reprogrammed and all nodes reboot from image 1.

3.1.3 Handling incremental network deployment

This approach works well when new nodes join the network. Let nodes $n_1, n_2, ..., n_k$ ($k \geq 1$) having an older version of application as image 1 and running Stream-

RS (image 0) join the network. Nodes $n_1, n_2, ..., n_k$ advertise the data they have using Stream-RS. When neighbors of nodes $n_1, n_2, ..., n_k$ running image 1 hear the advertisement, they reboot from their image 0 (Stream-RS). Note that the neighbors of $n_1, n_2, ..., n_k$ do not broadcast the reboot message and thus only the neighbors of $n_1, n_2, ..., n_k$ reboot from Stream-RS. For this, the nodes should be able to distinguish whether the reboot message is coming from the base node or non-base node. This is achieved by looking at the source field of the reboot message. We assume that all nodes in the network know the id of the base node. Now using the steps 2 through 6, the new nodes download the new application as image 1 and all nodes reboot from image 1 (the new application).

3.1.4 Design of Stream-AS

Stream-AS should be designed such that the increase in the size of the application image when it is attached to the user application is minimum and at the same time, the network should be able to reprogram itself whenever required. Stream-AS provides the functionality to reboot from image 0 when the user gives the reboot command. Before injecting the application to the network, user gives a reboot command to the base node. The base node running image 1 (user application plus Stream-AS) broadcasts the reboot command and itself reboots from image 0 (Stream-RS). The reboot command is flooded through the network. Ultimately all nodes in the network reboot from image 0 and actual application image transfer is done by Stream-RS. This kind of flooding technique used to reboot all the nodes in the network does not cause congestion because each node broadcasts the reboot command only once and reboots from Stream-RS immediately after.

Stream-AS provides functionality to reboot from image 0 when new nodes are introduced to the network. When new nodes join the network, they periodically broadcast the advertisement. After one hop neighbors of these new nodes hear the advertisement, they reboot from image 0 (Stream-RS). Then Stream-RS takes care

of sending the new application image to the new nodes. One disadvantage of the current implementation of Stream is that the new nodes must be running image 0 so that upon hearing the advertisement from these nodes, already deployed nodes can reboot from Stream-RS.

From the above discussion, it is clear that incorporating Stream-AS requires minimal change in the user application. In TinyOS, following is the nesC code required to be added when Deluge is attached to the user application:

Components DelugeC;

Main.StdControl→DelugeC;

To attach user application to Stream-AS instead, replace *DelugeC* by *StreamASC*.

This difference translates to a considerable difference in the size of the program image that is transferred over the wireless channel.

Steady-state Behavior

In Deluge, once a node's reprogramming is over, it keeps on advertising the code image that it has. This is to ensure that the new nodes joining the network get the latest version of the application image and also for future injection of code image. As a result, radio resources are continuously used by Deluge even in the steady state. On the other hand, in Stream, there is no advertising of data in the steady state because Stream-AS does not advertise the data that it has. As mentioned earlier, the nodes running user application plus Stream-AS in the steady state receive the advertisement from the new nodes running Stream-RS, reboot from Stream-RS and send the new application to the new nodes. When reprogramming is to be done, the nodes running user application plus Stream-AS get the reboot command from the base node, reboot from Stream-RS, and download the new application, thereby avoiding the need to advertise at steady state. That means the user application has to share the node's radio resources with Deluge while this is not the case when Stream is used. Also, since the nodes always run the user application except during reprogramming period,

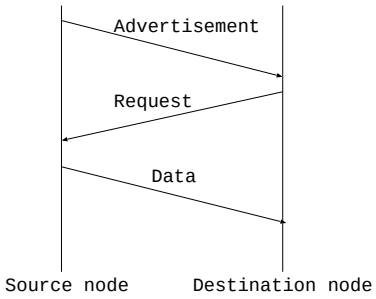


Fig. 3.1. Three-way handshake for data dissemination

RAM usage is much less for Stream than for Deluge because of the smaller size of the user application plus Stream-AS compared to user application plus Deluge.

3.1.5 Design of Stream-RS

Stream-RS, preinstalled in all nodes as image 0 and executed only during reprogramming phase, is responsible for actual image transfer among the nodes in the network. It is based on Deluge with the significant changes mentioned below. When a new application image is to be injected into the network, all nodes reboot from Stream-RS. Then, reprogramming is done by using a three-way handshake (Figure 3.1) in which each node broadcasts the advertisement about the code pages that it currently has. A node, upon hearing the advertisement of newer data than it currently has, sends a request to the node advertising newer data. The advertising node then broadcasts the requested code pages. Deluge optimizes this reprogramming method by a proper choice of the time when advertisements and code pages are sent. For a complete discussion of Deluge, see [92].

Changes from Deluge

Once reprogramming is done we want all the nodes to reboot from the new application automatically. One obvious approach would be to reboot each node from the user application after it completes downloading the new application. But the

flaw with this approach is that even though a node has completed downloading the new application, it may still be serving other nodes in the network. Therefore the node needs to continue to run Stream-RS. When a node receives a request for data, it puts the node-id of the requesting node in the set S. This set S is not shared and is maintained by each node. This structure is essential because a design philosophy of Stream is to have all nodes running the user application all the time except when reprogramming is being done. Without knowledge of the nodes to which a node A is currently sending code, A may reboot from image 1 after it has downloaded all the pages of the new user application even though some nodes in the network may still be receiving code fragments from A.

In Stream-RS when a node downloads the new application, it broadcasts an ACK saying that it has completed downloading the new application. When a node receives an ACK from its neighbor, it removes the id of that node from the set S. So, the following invariant is maintained at all times:

$$A.S = \{x | REQ(x, A) = true \wedge ACK(x, A) = false\}$$

This ensures that the set S at a node A consists of the ids of those nodes to which it is currently sending code fragments. The condition for a node A to reboot from the user application (image 1) is as follows:

$$A.S = \phi \wedge A.\sharp\, pages = Total\ number\ of\ pages$$

The first condition is that A is not sending code to any node and the second condition is that A itself has downloaded all the pages of the application. Eventually all nodes in the network download all the pages of the new application and reboot from image 1. So in the steady state all the nodes run the application attached with Stream-AS.

There is a subtle drawback to the synchronization in Stream-RS. A node A may be serving a node B without B having explicitly sent a request to A. Thus node B would never be included in A's set S. Let us consider a scenario when a node n_1

hears an advertisement of newer data than it currently has from node n_2 during the reprogramming phase. Before node n_1 sends request for the new data, some other one-hop neighbor n_3 of the advertising node n_2 may send the request. In response to the request from node n_3, n_2 broadcasts the code. So, the node n_1 may never send the request to node n_2 but keep on receiving the code from node n_2, triggered by request from node n_3. If all the nodes that explicitly request data from the advertising node n_2 complete downloading the new application earlier than the node n_1, node n_2 will reboot from the new application. This leaves node n_1 in the middle of downloading the new application. This drawback, however, does not pose a correctness problem, but a performance problem. This is due to the design that after the advertising node n_2 has rebooted from the user application, it still can hear the advertisement sent by the node n_1 due to Stream-AS. Upon hearing the advertisement, node n_2 will reboot from Stream-RS and start sending code to node n_1 through the three-way handshake.

In Deluge, in contrast to the automatic operation here, once all nodes complete downloading the new user application, they reboot from the new application only after the user gives the reboot command manually from the computer attached to the base node.

3.2 Stream Analysis

Here we present an approximate analysis of the reprogramming time and energy cost of uploading applications using three different protocols: Deluge, Stream and an ideal protocol in which only the application needs to be uploaded without any extra overhead. Let the application consist of N_p pages and each page has A_{pkt} packets. Let P_s be the probability of successful transmission of a packet over a single link.

3.2.1 Reprogramming time

In this section, we analyze the reprogramming time for a grid network. We assume that the transmission range is equal to $\sqrt{2}$ times the grid spacing. The reprogramming

model that we use for the analysis is an approximation of the behavior of Stream. In it, we divide the time line into fixed-size rounds. The source sends the advertisement at the beginning of each round and the destination, the one hop neighbor of the source that hears the advertisement, sends one request for each new advertisement received. We assume, for tractability of analysis, that the advertisement and the request packets are reliably delivered. This can be achieved in practice by either having a separate control channel or by transmitting the control signals multiple times to give a desired reliability. If this assumption is not true, then the multi-hop reprogramming time we find is a lower bound rather than the exact time. Once the source receives the request, the data packets are sent immediately. If not all the data packets in a page can be sent to the destination, the remaining data packets are sent over the following one or more rounds. The time T_r is defined as the time to send a new advertisement, receive a request, and send all the N_{pkt} packets of the page being advertised when the link reliability is 1.0. The number of rounds that it takes for all the packets in a page to be received at the destination is thus a random variable. Let us call it N_r. The probability of completing the upload of the entire page within the k^{th} round since the start of transmitting the page is the probability that each packet in the page is successfully delivered within k rounds. Assuming independence of the losses of different packets within a page, we have

$$P(N_r \leq k) = \left[\sum_{j=1}^{k} P_s(1-P_s)^{j-1}\right]^{N_{pkt}} \tag{3.1}$$

The expected number of rounds for successfully sending a whole page is

$$E[N_r] = \sum_{i=1}^{\infty} i.P(N_r = i) = \sum_{i=1}^{\infty} P(N_r \geq i) \tag{3.2}$$

$$E[N_r] = \sum_{i=1}^{\infty} (1 - P(N_r < i)) = \sum_{i=1}^{\infty} (1 - P(N_r \leq i-1)) \tag{3.3}$$

$$E[N_r] = \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right] \tag{3.4}$$

The code transmission is pipelined. That is, a node does not have to completely download the new image before sending it to the next hop. As soon as the node downloads the first page of the new application, it can send that page to the other nodes if it gets the request for that page. Since the page transmission is pipelined, the expected number of rounds it takes to download the whole application at a node $h-$hop away is given by,

$$E[N_{r,h}] = min\left\{3(N_p - 1) + h, N_P.h\right\} E[N_r] \tag{3.5}$$

Here $h.E[N_r]$ is the number of rounds to download the first page, $3.(N_p - 1).E[N_r]$ is the number of rounds to download the rest of the pages if the network spans across more than 4 hops because of two-hop interference effect on pipelining, i.e. at any point of time, if a node at hop h receives data from hop $h-1$, no node at hop $h+1$ can send data at the same time because of collision at hop h. For networks with maximum hop separation less than 4, there is no pipelining of the code transfer and $N_p.h.E[N_r]$ is the number of rounds to download all the pages. Plugging Equation 3.4 into Equation 3.5, we get

$$E[N_{r,h}] = min\left\{3(N_p - 1) + h, N_P.h\right\}.\sum_{i=1}^{\infty}\left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right] \tag{3.6}$$

Assuming maximum number of hops to be h_{max} and the round time to be T_r, the expected reprogramming time T_{rep} is

$$\begin{aligned} T_{rep} &= T_r.E[N_{r,h_{max}}] \\ &= T_r.min\left\{3(N_p - 1) + h, N_P.h\right\}.\sum_{i=1}^{\infty}\left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right] \end{aligned} \tag{3.7}$$

We calculate the reprogramming time for (a) standalone application (one that does not perform radio communication) and (b) application that uses *GenericComm* component (provided by TinyOS) for communication. The application size is taken to be 1, 10, or 100 pages. In case (a), the increases in the size of the program image (in units of a page) are 10 and 20 respectively for Stream and Deluge, while in case

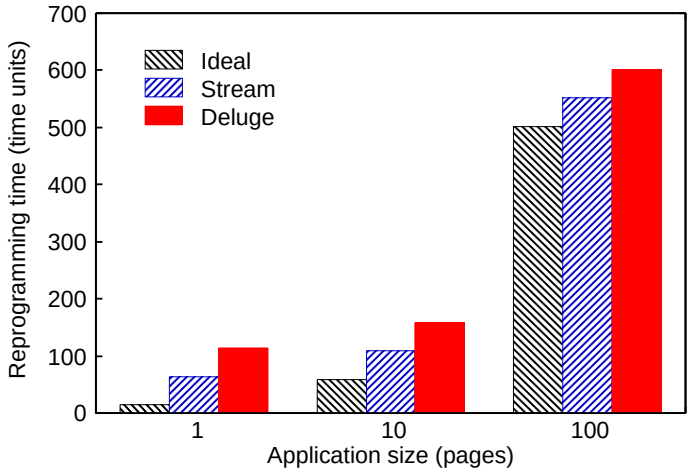


Fig. 3.2. Reprogramming time for 10x10 grid topology with standalone applications

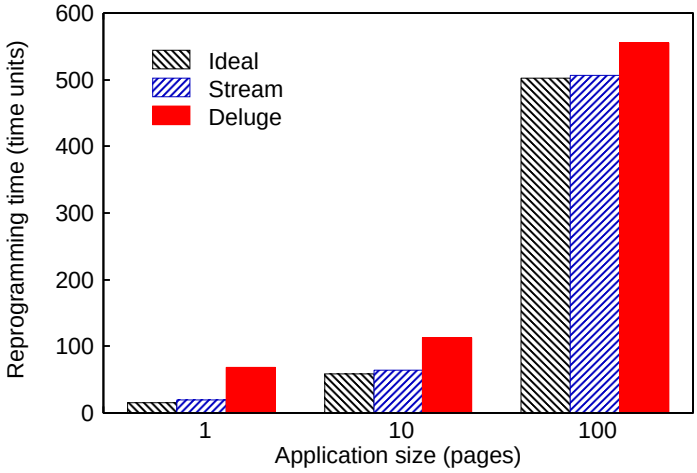


Fig. 3.3. Reprogramming time for 10x10 grid topology with applications having communication capability

(b), these increases are 1 and 11 respectively for Stream and Deluge. We use $P_s = 0.98$ and $T_r = 1$ time unit. Assuming that P_s stays constant across the three cases (Ideal, Stream, and Deluge), the reprogramming time becomes directly proportional to the number of pages (since the other factors are constant). Figure 3.2 and Figure 3.3 show the reprogramming time for the 10X10 grid. Clearly Stream outperforms

Deluge and for applications having communication features (almost all sensor network applications have this feature), Stream is almost as fast as the ideal case.

3.2.2 Energy Cost

Let C be the energy cost of transmitting a single packet. The energy cost of receiving packets depends on the specifics of the underlying application such as sleeping schedules. Packet transmission and writing received packets to external flash are the major sources of energy consumption during reprogramming. Both of these are functions of the number of packets transmitted during reprogramming. In this analysis, we count the number of transmitted packets and use that as an indicator of the energy performance of the two modes. This simplification is commonly done in the literature, e.g. [128]. Assuming that retransmissions of a packet are independent, the expected number of transmissions over a link for a successful transmission of a packet N_{ret} is

$$K = E[N_{ret}] = \sum_{k=1}^{k=\infty} \left[k.\left(P_s(1-P_s)^{k-1}\right)\right] = \frac{1}{P_s} \tag{3.8}$$

Let the *redundant set* at hop h be S_h, where S_h is the set of nodes at hop h that can be reprogrammed by one node at hop $h-1$. Let $|S_h|$ be the average size of the set. Moreover, let α_h be the cardinality of the subset of nodes at hop $h-1$ that can reprogram all the nodes at hop h. The additional energy cost to reprogram all the nodes at hop h given that all the nodes at hop $h-1$ have been reprogrammed is given by

$$E_h = K.N_P.N_{pkt}.C.\alpha_h = \frac{N_P.N_{pkt}.C.\alpha_h}{P_S^{|S_h|}} \tag{3.9}$$

The total energy overhead of reprogramming all the nodes in a network in which the maximum number of hops is h_{max} is given by

$$E = \sum_{h=1}^{h=h_{max}} E_h = \sum_{h=1}^{h=h_{max}} \left[\frac{N_P.N_{pkt}.C.\alpha_h}{P_S^{|S_h|}}\right] \tag{3.10}$$

For a linear topology of N nodes with $R_{tx} = d$, where d is the spacing between nodes, and R_{tx} is the transmission range, $\alpha_h = 1$, $|S_h| = 1$, and $h_{max} = (N-1)$. For an $n \times m$ grid topology, ignoring edge effects, with $r = \sqrt{2}d$, $\alpha_h = \lceil \frac{n}{2} \rceil$, $|S_h| = 3$, and $h_{max} = (m-1)$. Let $N_{pkt} = A_{pkt} + 1 + E[N_r]$, where the second term is to account for the advertisement packet and the last term represents the expected number of request packets to successfully transmit the whole page (Equation 3.4).

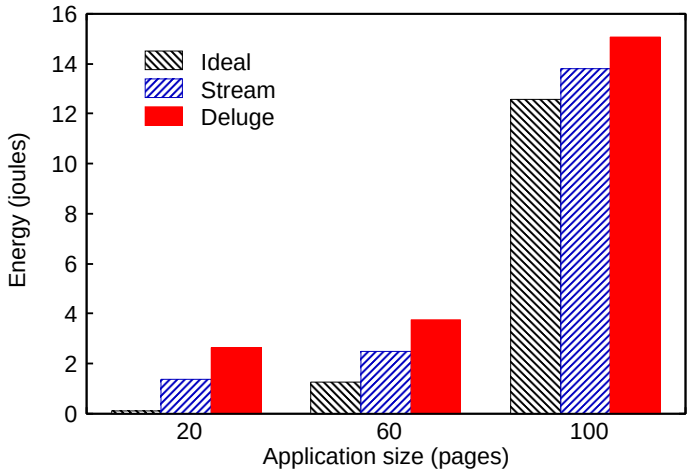


Fig. 3.4. Total energy consumed in the 10x10 grid topology with standalone applications

We calculate total energy E expended for (a) standalone application (one that does not perform radio communication) and (b) application that uses *GenericComm* component (provided by TinyOS) for communication. The application size is taken to be 1, 10, or 100 pages. In case (a), the increases in the size of the program image (in units of a page) are 10 and 20 respectively for Stream and Deluge, while in case (b), these increases are 1 and 11 respectively for Stream and Deluge. We use fixed energy cost as 50 nJ/bit, $P_s = 0.98$, the variable (distance dependent) energy cost is 100 pJ/bit $\times$ r^2, for a transmission distance of r, the receiving energy is equal to the fixed energy cost. As expected, Figure 3.4 and Figure 3.5 show that Stream is faster and more energy efficient than Deluge. These figures also reaffirm that for the communicating applications, energy costs of Stream and ideal case are comparable.

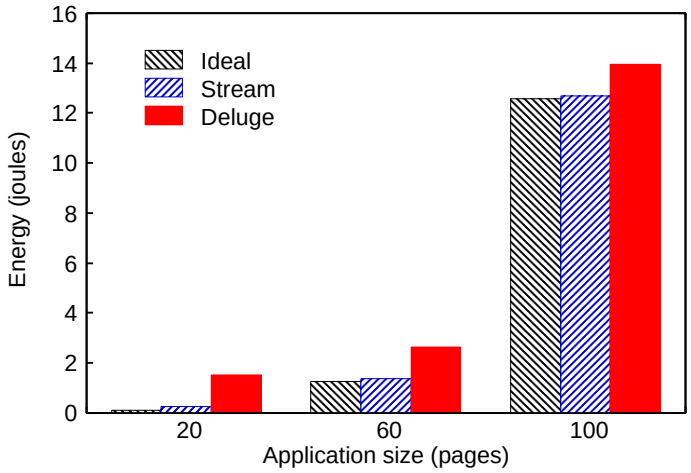


Fig. 3.5. Total energy consumed in the 10x10 grid topology with applications having communication capability

3.3 Experiments and Results

We implement Stream using the nesC [126] programming language in TinyOS [14]. In this section, we compare the performances of Stream and Deluge for different network sizes and node densities. Both testbed experiments and simulations are used to demonstrate the advantages of Stream over Deluge. Testbed experiments are performed by using Mica2 [1] nodes and simulations are performed using TOSSIM [127], a bit level simulator for TinyOS platform. Testbed experiments show the performance of Stream and Deluge in realistic environment while simulations exhibit the scalability of these protocols.

3.3.1 Evaluation Metrics

Any network reprogramming protocol must ensure that all nodes in the network receive the application image completely in a short period of time without expending too much energy. Reliability of code upload is an important evaluation metric. A second important metric is the time required to reprogram the network since the network functionality is degraded during reprogramming. Since the sensor network

consists of energy-constrained sensor nodes, the reprogramming protocol should use minimum energy to increase the lifetime of the network.

Both Deluge and Stream are 100% reliable, i.e. all nodes in the network download every byte of the user application. So, in the following sections, we focus on comparison in terms of time to reprogram the network and the energy consumed during reprogramming.

3.3.2 Testbed Description and Results

We perform the experiments using Mica2 nodes having a 7.37 MHz, 8 bit microcontroller. Each Mica2 node is equipped with 128KB of program memory, 4KB of RAM and 512KB external flash which is used for storing multiple code images. These nodes communicate via a 916 MHz radio transceiver.

The first set of experiments is performed in 2X2, 3X3 and 4X4 square grid networks having a distance of 10 ft between adjacent nodes in each row and column. Experiments of network reprogramming using Stream are carried out by pre-installing Stream-RS as image 0 and same version of application image plus Stream-AS as image 1 on all nodes in the network. A new application image plus Stream-AS is injected into the source node (situated at one corner of the grid) via a computer attached to it. Then the source node starts disseminating the new application image to the network. Experiments with Deluge are performed similarly by installing Deluge as image 0 and the application image plus Deluge as image 1. A new application image plus Deluge is injected into the network.

Time to reprogram the network is the time interval between the instant t_0 when the source node sends the first data packet to the instant t_1 when the last node (the one which takes the longest time to download the new application) completes downloading the new application. Since clocks maintained by the nodes in the network are not synchronized, we cannot take the difference between the time instant t_1 measured by the last node and t_0 measured by the source node. Although a synchronization

protocol can be used to solve this issue, we do not use it in our experiments because we do not want to add to the load in the network (due to synchronization messages) or the node (due to the synchronization protocol). Instead, once each node completes downloading the new application image, it sends a special packet to the source node saying that it has completed downloading the new application. The source node measures the time instant t_1' when it receives such packet, timestamps the packet with t_1' and sends the packet to the computer. If the network has n nodes including the source node, the computer attached to the source node receives one t_0 and $(n-1)$ number of t_1''s. We take $t_{prog} = \max_{t_1'}(t_1' - t_0)$ as the reprogramming time. It should be noted that the actual reprogramming time is $\max_{t_1'}(t_1' - t_0 - t_d)$ where t_d is the time required to send the special packet from the last node to the source node. Since t_d is negligible compared to the reprogramming time, our formula is a reasonable approximation to the actual reprogramming time. Furthermore, since we are interested in the difference between the reprogramming times of Stream and Deluge, the effect of t_d cancels out assuming t_d is same for Stream and Deluge experiments.

Among the various factors that contribute to the energy used in the process of reprogramming, two important ones are the amount of radio transmissions in the network and the number of flash-writes (the downloaded application is written to the external flash as image 1). Since the radio transmissions are the major sources of energy consumption, we take the total number of bytes transmitted by all nodes in the network as the measure of energy used in reprogramming. In our experiments, each node counts the number of bytes it transmits and logs that data to its external flash. By reading the external flash and taking the sum of the number of bytes transmitted by each node, we find the total number of bytes transmitted in the network for the purpose of reprogramming. Since the amount of flash-writes in Deluge is higher, the energy advantage will be increased if we take that factor into account.

As mentioned earlier, compared to Deluge the exact gain achieved by Stream in terms of number of pages transmitted depends on the user application. For a simple application which does not write to external flash and does not perform any radio

communication, attaching Deluge to the user application increases the size of the application image by 20 pages whereas the increase is only 10 pages with Stream. If the user application has radio communication functionality but does not write to external flash, Stream and Deluge increase the number of pages by 1 page and 11 pages respectively. In our experiments, we use a simple application that performs radio communication but does not write to external flash. The application image alone is 11 pages, application image plus Stream-AS is 12 pages and application image plus Deluge is 22 pages.

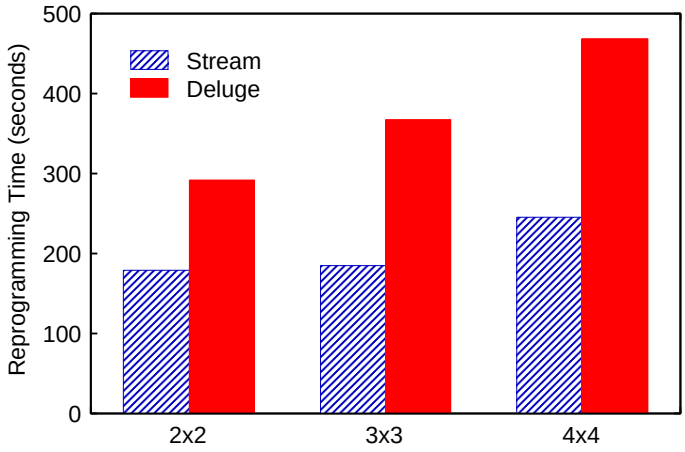


Fig. 3.6. Comparison of reprogramming time of Stream with that of Deluge for grid networks

Figure 3.6 compares the average time taken by Stream and Deluge to reprogram 2X2, 3X3 and 4X4 grid networks. Interestingly, we observe from the experiments that the number of hops between two nodes is dependent on environmental conditions and changes during multiple runs of the experiment. For example, a node is sometimes able to communicate with a node separated by more than one grid point. Expectedly, the experiments show that Stream reduces the reprogramming time significantly. This large gain in reprogramming time is because Stream needs to transfer only 12 pages whereas Deluge has to transfer 22 pages. The reduction in reprogramming time becomes more pronounced for larger networks. Figure 3.7 shows the total

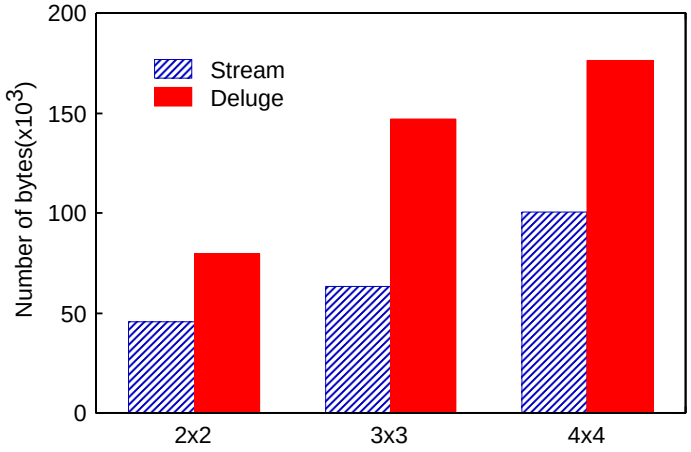


Fig. 3.7. Comparison of number of bytes transmitted in the network by Stream with that of Deluge for grid networks

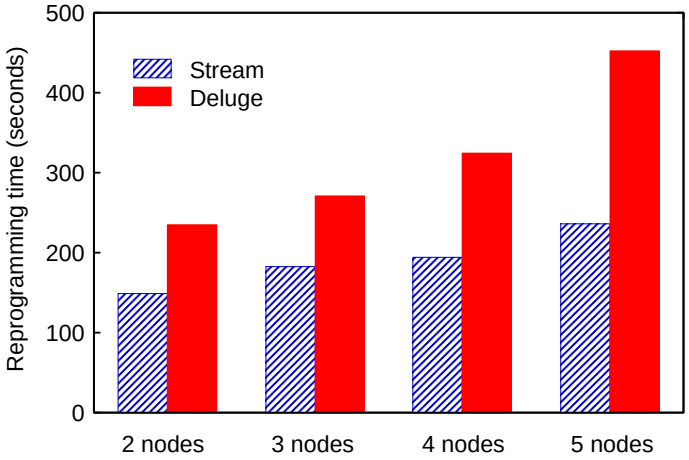


Fig. 3.8. Comparison of reprogramming time of Stream with that of Deluge for linear networks

number of bytes transmitted in the network during the reprogramming period. Both data packets and control packets (request and advertisement packets for Deluge and request, advertisement and ACK packets for Stream) are considered while calculating the number of bytes. These results indicate that the energy required to reprogram the network can be greatly reduced by using Stream instead of Deluge. Although the percentage gain in reprogramming time and total number of bytes transmitted in the

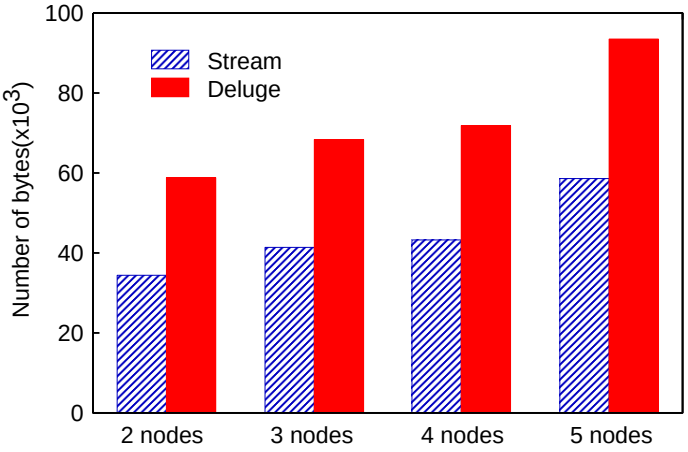


Fig. 3.9. Comparison of number of bytes transmitted in the network by Stream with that of Deluge for linear networks

network vary for different topologies, in our experiments we see that Deluge requires 63% to 98% more reprogramming time and transfers 75% to 132% more number of bytes than Stream for these grid topologies.

Next we performed the experiments for linear topologies with 10 ft. distance between adjacent nodes. Source node is situated at one end of the line. Figure 3.8 and Figure 3.9 provide the comparison of the reprogramming time and total number of bytes transmitted in the network respectively between Stream and Deluge for different sizes of the linear topologies. Again Stream is more efficient than Deluge with respect to reprogramming time and energy used for reprogramming. In our experiments, we noticed that compared to Stream, Deluge takes 58% to 90% more reprogramming time and transfers 59% to 70% more number of bytes for different linear topologies. If we compare the reprogramming time of 2X2 grid and linear network with 4 nodes, we find that the latter takes longer time to reprogram itself because 2X2 network can involve at most 2 hop communications (mostly 1 hop) while 4 linear nodes can have at most 3 hop communications.

The above graphs show only the number of bytes that are transmitted during the reprogramming period. In Deluge, each node keeps on broadcasting the advertisement

packets even after the reprogramming period is over. As a result, the nodes have to spend energy in advertising even when reprogramming is not being done. Stream does not have this problem because as soon as the reprogramming period is over, the nodes reboot from the application image plus Stream-AS which does not broadcast advertisements. As a result, we observe a monotonically increasing difference in the number of bytes as the protocols are allowed to continue to run in the steady state.

3.3.3 Simulation Results

In order to demonstrate the scalability of Stream and to compare it with Deluge for larger network sizes on the order of hundreds of nodes, we performed some simulations using TOSSIM [127], a discrete event simulator for TinyOS. Although TOSSIM does not model TinyOS hardware precisely, it provides more accurate modeling of the physical layer than many other simulators, such as ns-2. As TOSSIM does not model execution time accurately, the simulation results presented here only exhibit the overall behavior and trend and proper scaling is required to give the absolute values for the Mica2 platform. Since it takes tens of hours to complete simulations for larger networks, in our simulations, we reduce the number of packets per page from 48 to 24 packets. This reduction in page size is not of serious concern because we are interested in the comparison of performances of Stream and Deluge and not on the absolute values.

Effect of Network Size

We use several square grid networks (10 ft distance between successive nodes in any row and column) of varying size (up to 16X16 grid) for our simulations. A source node at one corner of the grid disseminates the user application to all other nodes in the network. Like before, Stream and Deluge need to transfer 12 and 22 pages respectively to all nodes in the network. Figure 3.10 and Figure 3.11 compare the reprogramming times and number of bytes transmitted in the network between

Stream and Deluge for different grid sizes. It shows that both Stream and Deluge are scalable, at least up to 256 nodes simulated. In our experiments, we found that compared to Stream, Deluge requires 41% to 101% more reprogramming time for different network sizes. We noticed that the increase in the total number of bytes transmitted in the network for Deluge compared to Stream was between 75% to 112% for different network sizes.

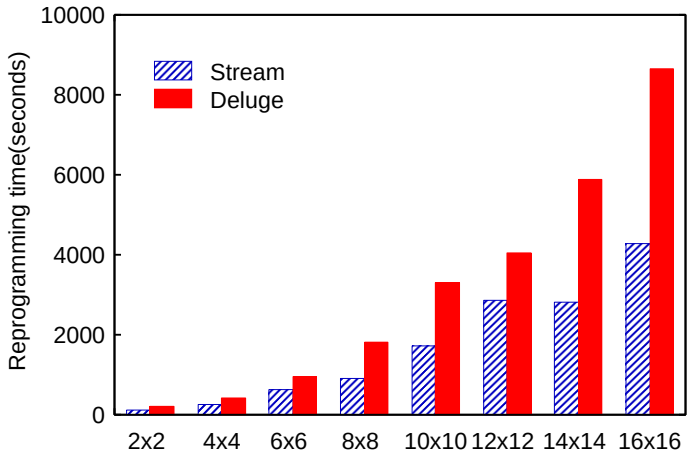


Fig. 3.10. Simulation comparison of reprogramming time of Stream with that of Deluge for nxn grid networks

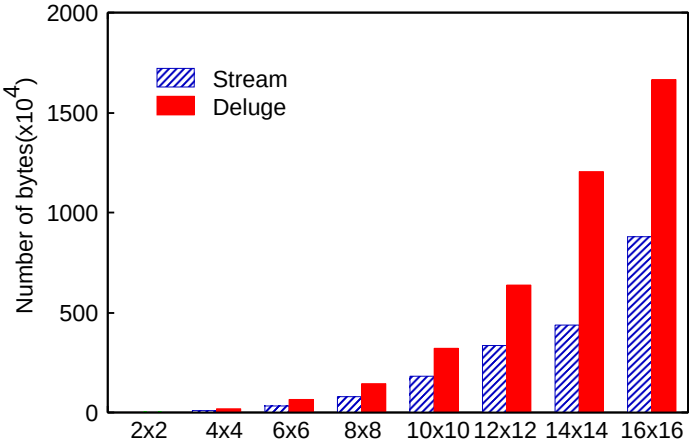


Fig. 3.11. Simulation comparison of number of bytes transmitted by Stream with that by Deluge for nxn grid networks

The goals of the mathematical analysis in Section 3.2 and simulation in this section are not to find the exact values of the reprogramming time and energy. The exact mathematical analysis is extremely difficult to do and as far as we know, there has been no such attempt in any published literature. Also TOSSIM does not simulate the node hardware and execution time exactly. So our objective in both mathematical analysis and simulation is to compare Stream and Deluge in terms of reprogramming time and energy and show the trend of these quantities as a function of the parameters like network sizes, size of the user application, etc. For example, if we compare the ratios of the reprogramming time for Deluge to that for Stream for 10X10 grid networks, it is about 1.8 from mathematical analysis and about 1.9 from simulation results. The results from simulation and analysis can be used only to observe the trend and to compare the two protocols. It would be incorrect to try to obtain accurate absolute values of any of the output parameters from either simulation results or analytical results. The absolute results are given by the testbed experiments.

Effect of Network Densities

To compare the performances of Stream and Deluge for different node densities, we vary the number of nodes in a 90 ft by 90 ft area. For each node density, the nodes are still arranged in grid fashion with uniform spacing between the adjacent nodes (just the spacing decreases with increasing density). Figure 3.12 shows that Stream reprograms the network much faster than Deluge for all network densities and Figure 3.13 shows that Stream uses lesser number of bytes than Deluge. The increase in node density increases the reprogramming time due to two reasons. First, there is an increase in the number of nodes in a given area resulting in more collisions of the transmitted packets. Second, there are simply more nodes that need to download the new application. These figures show that for higher node densities, the gap between reprogramming times as well as number of bytes between Stream and Deluge

widens further. This can be explained by the fact that Stream reduces collisions more effectively due to the reduced number of bytes transferred.

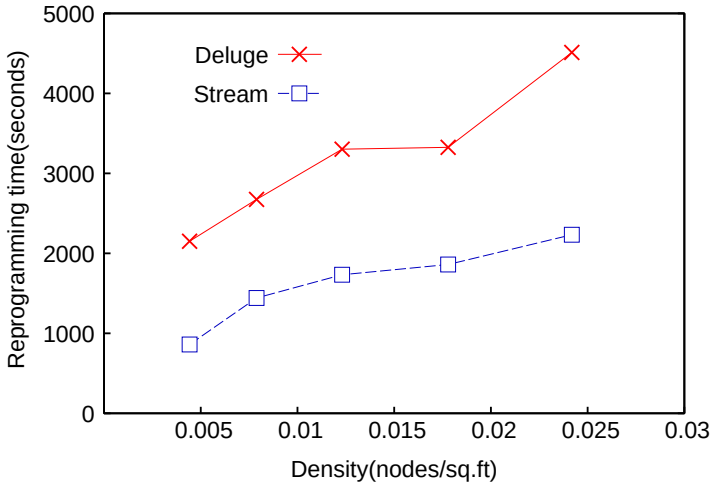


Fig. 3.12. Comparison of reprogramming time of Stream with that of Deluge for different node densities

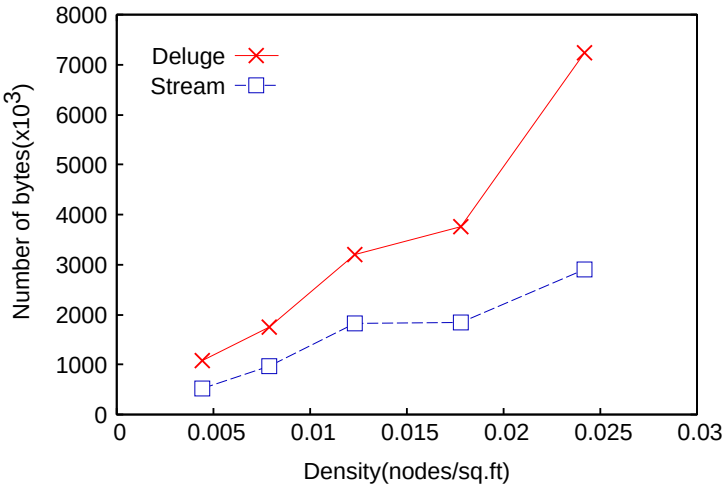


Fig. 3.13. Comparison of number of bytes transmitted by Stream with that by Deluge for different node densities

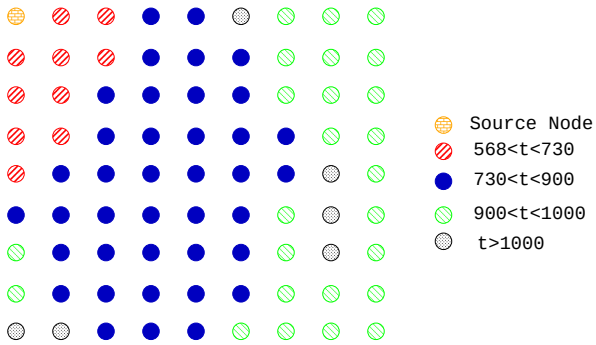


Fig. 3.14. Code dissemination profile according to the convergence time of a node

Profile of Code Dissemination

Figure 3.14 shows the profile of code dissemination with Stream in a 9X9 grid with 10 ft separation. The fill-pattern of the node indicates its time to download the application code. The results indicate that the dissemination takes place uniformly with hop distance from the source (which is at the top left corner). The results are close to what we get for Deluge and matches with what the authors find in [92] for a low density network.

3.4 Stream with Opportunistic Node Sleeping

3.4.1 Background and Rationale

Since sensor networks consist of energy-constrained nodes, one of the biggest goals of any sensor network protocol is to minimize the energy consumed by the nodes. As idle listening is a major source of energy consumption, many practical sensor network applications put the nodes to sleep as much as possible when the nodes are not doing any work, primarily sensing and communicating. Thus the sensor network reprogramming protocol should also aim at putting the nodes to sleep mode whenever they are not actually involved in reprogramming. When the new code is injected into the network, it takes some time for the new code to reach the nodes which are far

from the point of injection in the network. Freshet [102] cleverly estimates how long it takes for the new code to reach one of the neighbors of a given node and puts that node to sleep for the estimated duration. It does this by having each node estimate the number of hops it is away from the point of injection and then using an empirically calculated formula relating the time to the number of hops. Secondly, Freshet also puts the node to sleep for part of the time in the quiescent phase, i.e., when the code upload is complete. Note that Stream possesses the second advantage because immediately after the code upload is over, all nodes in the network reboot from image 1 (User application + Stream-AS) and the user application can put the node to sleep as it deems necessary. If Stream is operated in the alternate mode (to be described in this section), it can also achieve the other advantage of Freshet—putting nodes to sleep until the newly injected code arrives at one of its neighbors.

In this new mode of operation called Opportunistic Sleeping, Stream does not reboot a node from image 0 (Stream-RS) until the new code arrives at one of the neighbors of the node. A sensor node running image 1 (User application + Stream-AS) reboots from image 0 for reprogramming only if it hears an advertisement which is different from its metadata. We will describe this mode in Section 3.4.2. It should be noted that in Freshet, each node estimates how long it takes for the code to reach one of its neighbors based on its distance (hop count) to the base node. Because of the variability of the code propagation characteristics, such an estimate may not always be accurate resulting either in decrease of energy savings (underestimate of the time for the code to reach the node) or delay in reprogramming (overestimate). However, Opportunistic Sleeping mode of Stream does not have this problem because it does not put the node to sleep for some estimated interval, but it causes the node to reboot from image 0 (Stream-RS) only when the new code actually arrives at one of its neighbors.

Some sensor network applications may require all the nodes in the network to be running the same version of the application at any given time. The opportunistic

sleep mode of Streams operation violates this requirement. As a result, this mode should be used only if such restrictions are not present.

3.4.2 Protocol Description

Consider that initially nodes in the network have Stream-RS as image 0 and Stream-AS plus user application as image 1. A new user application attached to Stream-AS is injected at the base node. Then the base node running Stream-RS (image 0) is placed within the communication range of its single hop neighboring nodes. All nodes in the network are reprogrammed with the new code as follows:

1. Let α_1 be the set of nodes running image 1 which are within the communication range of the base node. Each of these nodes listens to the advertisement from the base node. The advertisement as usual carries an image version number, total number of pages in the code image and number of complete pages. If the advertisement is different than the code image version it currently has, each node in α_1 reboots from image 0. Otherwise, it does not reboot from image 0. Since the base node has newly injected image 1, all nodes in α_1 reboot from image 0. Note that unlike in the primary mode of Stream, no reboot message needs to be broadcast through the network.

2. After nodes in α_1 reboot from image 0, they start getting pages of the new code image from the base node. The three-way handshake to get the code image pages remains unchanged from the primary mode of Stream.

3. The nodes in α_1 advertise their metadata once they have a completed page of the code image. In line with existing multi-hop reprogramming protocols, they do not wait for the entire code image to be downloaded before advertising their metadata. This is identical to the primary mode of Stream.

4. Let α_2 be the set of nodes which are within communication range of at least one node in α_1. When nodes in α_2 hear the new advertisement from nodes in

α_1, they also reboot from image 0. In this way, the nodes reboot from image 0 for the purpose of reprogramming only when the new code image has arrived at one of its neighbors.

5. Using the same method as explained in 3.1.2, all nodes in the network get reprogrammed and reboot from image 1 (the new code image that was just downloaded).

A node running image 1 (Stream-AS plus user application) has to check the advertisement received from another node against its own metadata, consisting of information about the images it currently has, such as version number. It uses this information to decide if it has to reboot from Stream-RS for reprogramming. Reading data from the external flash where the metadata is stored is expensive because it increases the size of image 1 (Stream-AS plus user application) and also incurs a higher current draw. The exact amount of increase in image size is application dependent, but for the user applications which do not use external flash, this increase is 2 pages. Since the goal of Stream is to reduce the size of image 1 that gets transferred over the wireless medium during reprogramming, this increase in size reduces the advantage of Stream. To avoid this, Stream stores the metadata information in internal EEPROM memory. Reading from internal EEPROM increases the size of image 1 by few bytes (about 50 bytes in the applications that we considered). So, this mode of operation does not sacrifice the gains of original Stream.

In opportunistic sleeping mode, no reboot message is explicitly sent. Instead, a node x running Stream-AS decides whether to reboot from Stream-RS based on the advertisement message it receives from say node y. If the advertisement from the node y has $y.version \neq x.version$, then the node x reboots from Stream-RS. If $y.version < x.version$, then the node x reboots with the intention of bringing the node y up-to-date. However the neighbors of node x (z_1, z_2, z_3 in Figure 3.15) will not reboot from Stream-RS if they are not neighbors of node y because they only hear the advertisement from node x and $x.version = z1.version = z2.version = z3.version$.

On the other hand, if $y.version > x.version$, node x will reboot from Stream-RS and once it gets the first page of the new image, it will start advertising the new image. Then the neighbors of node x ($z1, z2, z3$ in Figure 3.15) will also reboot from Stream-RS because they find that node x has new version of the image. As a result, those nodes also start getting reprogrammed with the new image.

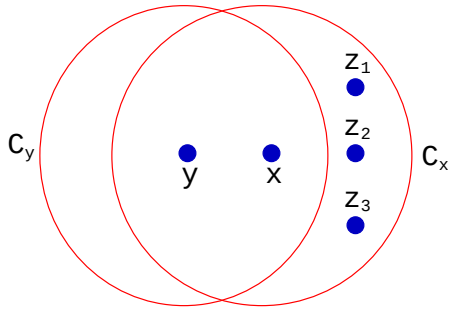


Fig. 3.15. Illustration of which nodes reboot for reprogramming. C_x and C_y are the communication ranges of nodes x and y respectively.

3.4.3 Experiments and Results

We performed testbed experiments and TOSSIM simulations with opportunistic sleeping mode of Stream to show how long a node can sleep before rebooting from image 0 during the initial stage of reprogramming before the newly injected code arrives at one of its neighbors. Note that the delay between the injection of the new code image and the instant when this code arrives at one of its neighbors of a given node depends on how far (hop count) the node is from the base node. As this hop distance increases, the delay also increases. The advantage of this mode of Stream is pronounced only for larger-hop networks. To do this with the limited number of sensor nodes that we have, we ran the experiments on various linear topologies having up to 11 nodes. As shown in Figure 3.16 an originator node (node 0) situated at one end of the line disseminates code to all the nodes in the network. Let the nodes be arranged as shown in Figure 3.16 where the node next to node 0 is node 1, the node

next to node 1 is node 2 and so on. To achieve maximum possible hops between the base node and the farthest node from the base node, we restrict the communication of a node i with node $(i-1)$ and node $(i+1)$ only. Figure 3.17 shows the delay (time interval between the instant when new code is injected to the base node to the instant when the node reboots from image 0) for each node in the network as a function of its hop count from the base node. As expected, the delay increases with the hop count. Note that significant amount of energy can be saved if the nodes sleep for these intervals since the current draw in the sleep mode is three orders of magnitude less than in the idle mode for the mote class of sensor nodes (μA versus mA).

As explained above, the advantage of the new mode of Stream becomes more pronounced for larger-hop networks. So, we conducted TOSSIM simulations for linear networks of sizes up to 150 nodes. As with testbed experiments, we restrict the communication of a node i with node $(i-1)$ and node $(i+1)$ only to achieve maximum possible number of hops in the network. Figure 3.18 shows the simulation results. The amount of energy saved by the opportunistic sleeping mode of Stream is quite significant for larger networks. Note that TOSSIM simulations do not provide the exact numbers and should be used only to observe the trend.

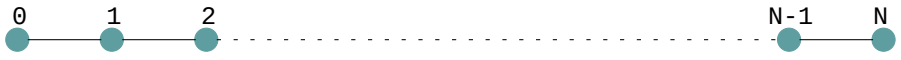


Fig. 3.16. Linear topology with nodes being reprogrammed using alternate mode of Stream with node 0 as the base node (N=1,2,...,11)

It should be noted that in real sensor network applications, the exact amount of time a node sleeps depends on the sleep cycle schedule of the user application running on the sensor node. Stream cannot unilaterally decide how long a node can sleep. It can put the node to sleep only during those time intervals when the user application running on the node does not require it to be awake. Without this new mode of Stream, nodes reboot from Stream-RS and remain awake even during the delay for the code to reach its vicinity. But with the new mode of Stream, the nodes

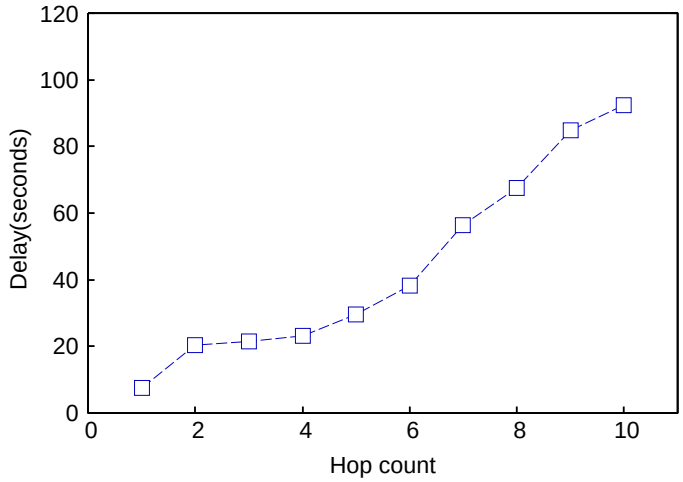


Fig. 3.17. Testbed result: Delay before a node reboots from Stream-RS for reprogramming as a function of its hop count from the base node

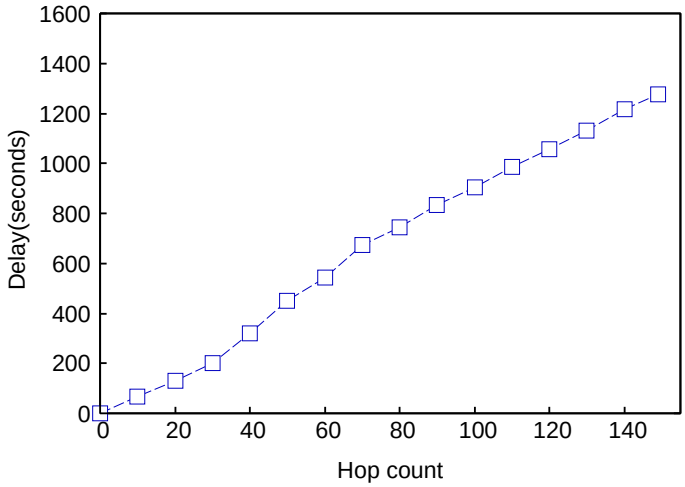


Fig. 3.18. Simulation result: Delay before a node reboots from Stream-RS for reprogramming as a function of its hop count from the base node

can sleep during this delay interval if the user application does not require it to be awake. In other words, depending upon the user applications sleep schedule, the nodes may not sleep for the entire delay interval shown in Figure 3.17 and Figure 3.18. Since most of the sensor applications sleep for a longer interval before waking

up for a short interval to do its job (sensing, sending data etc), the sleep time will likely be close to the delay values shown in Figure 3.17 and Figure 3.18. The energy savings per reprogramming due to this delay is shown in Figure 3.19 for different duty cycles (ratio of sleep time to one cycle consisting of sleep time and awake time) of the sensor node. Energy savings is calculated using the formula: Savings = Voltage $\times$ (Current for idle radio + Current for idle CPU) $\times$ Sleeping time (as calculated from the TOSSIM experiments) $\times$ (1-duty cycle). For mica2 platform, current for idle radio=7.03 mA, current for idle CPU=3.2 mA. We neglect the current draw in the sleep mode, which is less than 1 μA.

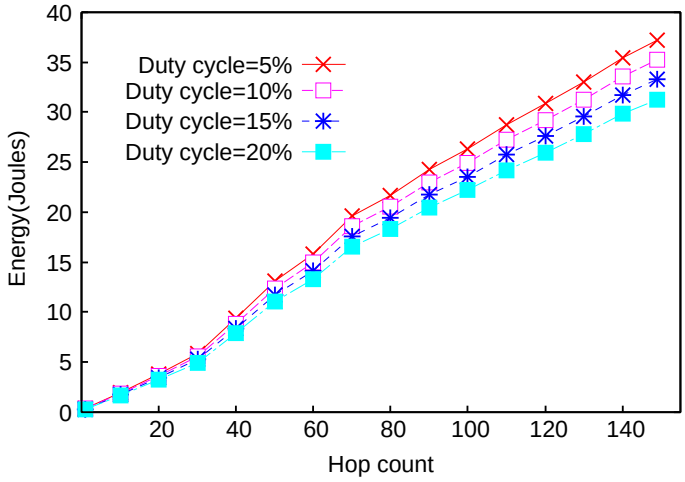


Fig. 3.19. Energy saving achieved by Stream over Deluge due to nodes sleeping till the code image arrives at its vicinity

3.4.4 Mathematical Analysis

Next, we analyze the delay between the injection of the code in the base node and the instant when it arrives at one of the neighbors of a node h hops away from the base node. For a node one hop away from the originator of the new code, this time interval is the time for a single round of a three way handshake. Assuming perfect pipelining of the single page of the code, the time interval $T_{delay,h}$ for a node

h hops away from the originator of the new code is $T_{delay,h} = h.T_{round}$ where T_{round} is the time for a single round of the three way handshake. T_{round} consists of following components:

$$T_{round} = T_{adv} + T_{req} + T_{data}$$

where T_{adv} is the time used by the nodes in advertising their metadata before the node requiring the new code decides to send the request. T_{req} is the time used for requesting the data and T_{data} is the time required to send one page of data.

To calculate T_{adv}, T_{req} and T_{data}, we need to find the expected number of transmissions required for a successful transmission of a packet. Let P_s be the probability of a successful transmission of a packet over a single hop. Assuming that the retransmissions of a packet are independent, the probability that the number of transmissions of a packet, N_{tx}, equals k is given by

$$P(N_{tx} = k) = (1 - P_S)^{k-1} P_S$$

The expected number of transmissions for a given packet is

$$E[N_{tx}] = \sum_{k=1}^{\infty} k(1 - P_S)^{k-1} P_S = \frac{1}{P_S}$$

T_{adv} can be approximated as follows:

$$T_{adv} = E[N_{tx}](t_l + GX^2 + T_x + T_{proc})$$

where t_l is the approximate time interval between two advertisements. Note that Stream uses Deluges advertisement policy. It divides time into intervals $[t_l, t_h]$ and each node decides whether to advertise or not in a given interval based on the number of similar advertisements it has heard in the previous interval. We take the lower value t_l because once the originator gets the new version of the code, it sets its advertisement period to t_l and the nodes hearing the advertisement from the originator also set their advertisement periods to t_l. We also assume that there were not enough similar advertisements in the previous interval to prevent the node from advertising in the current interval. GX^2 is the MAC delay for a single packet, where X is the number

of contending nodes. The MAC delay is difficult to compute analytically and to the best of our knowledge, no closed form solution has yet been proposed. The authors in [129] show that for the region of interest (low contention) the delay is approximately proportional to the square of the number of contending nodes. Here G is the proportionality constant and X is the number of contending nodes. T_x is the transmission time for a single packet. T_{proc} is the processing time required by a node after receiving the packet. T_{req} can be calculated as follows:

$$T_{req} = E[N_{tx}]E[N_{reqs}]\left(E[t_r] + GX^2 + T_x + T_{proc}\right)$$

where N is the number of packets in a page.

Figure 3.20 shows the delay as a function of hop count for a grid network with δ=10 ft separation between adjacent nodes. Probability of successful transmission of a packet P_s is taken as 0.9. For Stream, t_l=2 seconds, t_r=0.5 seconds and N=48 packets. From [92], we take $E[N_{reqs}]$=5.4. For mica2 node, transmission rate is 19.2 Kbps and hence T_x=0.015 seconds. We take T_{proc}=0.001 seconds. To calculate MAC delay GX^2, we take G=1 [105]. For a given node, the number of contending nodes varies with the location of the node in the network. For example, for the grid network, the nodes along the diagonal of the grid have higher number of contending nodes while those at the periphery have less contending nodes. We assume that the network is large and hence the average number of contending nodes is $9/4\delta^2$ (eliminating boundary effects) and the number of contending nodes is $9/4\delta^2 \times \pi r^2$ where r is the transmission range. The interference range of a node may be different from its transmission range. The difference can be easily accommodated in our analysis by replacing the communication range with the given interference range. Figure 3.20 shows that the time delay before the node reboots from Stream-RS for reprogramming is quite large for the nodes distant from the originator of the code. Under the new mode of operation of Stream, the nodes can sleep for this duration, and thus Stream can conserve energy for network reprogramming which increases with the network size. As mentioned before, results from TOSSIM simulations and mathematical analysis are not exact and should be used just to observe the trends. The trends of the delay values with increasing hop

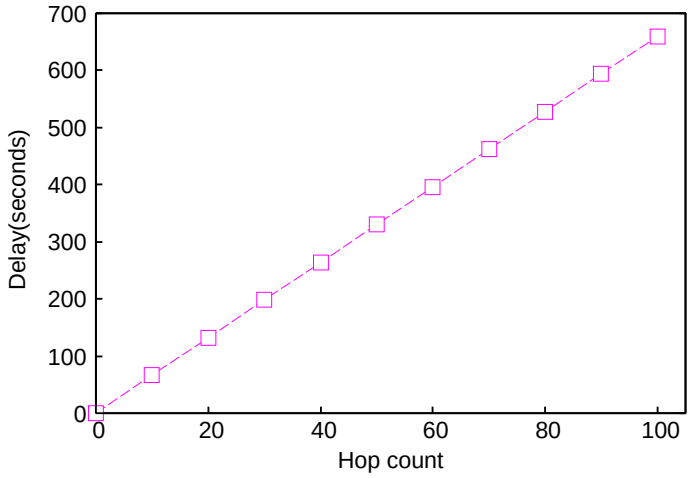


Fig. 3.20. Analytical result: Delay before a node reboots from Stream-RS for reprogramming as a function of its hop count from the base node

count obtained from analysis (Figure 3.20), real testbed experiments (Figure 3.17) and simulations (Figure 3.18) are comparable.

3.5 Effect of User Application's Sleep-Awake Scheme on Reprogramming

We have to be aware that Stream does not execute in isolation at the sensor nodes. The nodes run some user application which generally causes the node to operate with a low duty cycle, i.e., the node sleeps for most of the time and wakes up for short time interval to perform its tasks (like sensing, sending data to base station, etc.). This helps the node to reduce the energy consumption due to idle listening and thus to lengthen the lifetime of the node. Generally, for the best performance (small reprogramming time), all nodes in the network should be awake during the reprogramming period. So, for the quick reprogramming of the network, it is better if the network owner puts all nodes to the awake state during reprogramming. If that is not the case, all the existing reprogramming protocols (Deluge, Freshet etc) including Stream (both the original and the opportunistic sleeping modes) suffer performance degradation, i.e., they take longer time to be reprogrammed.

Beside performance degradation, the reprogramming protocol may not be able to reprogram all the nodes in the network due to the application induced sleeping of the nodes. For example, in Deluge, when all nodes complete downloading the new image, the user has to manually send a reboot command to reboot all the nodes in the network from the freshly injected image. If at the time when the reboot command is issued by the user, some nodes are not awake, they cannot reboot from the new image. The same problem occurs in Freshet also. In Stream (original mode), this problem is slightly different. Although the nodes automatically reboot from the newly injected image (after it is completely downloaded), the user has to give the reboot command to reboot all the nodes in the network from Stream-RS while starting the reprogramming process itself. Therefore all nodes need to be awake at that time. However, in the opportunistic sleeping mode of Stream, this problem does not exist. The user does not have to give the reboot command at all. But another problem can occur in the alternate mode of Stream. Let us assume that n_1 and n_2 are respectively the first hop and second hop neighbors of the base node. Let the base node has new image stored in its external Flash and it is running Stream-RS. The node n_1, running Stream-AS plus user application, hears the advertisement from the base node and since the base node's advertisement says that it has a new image, n_1 reboots from Stream-RS and starts downloading the pages of the new image from the base node. Let us assume that n_2 never wakes up during the entire period when n_1 is downloading the new image and sending out its advertisements. Then n_1 reboots from the new image since there was no request to it for the new image. As a result, n_2 is not reprogrammed. To overcome this possible problem, each node running Stream-AS plus user application periodically sends the advertisement so that even if a node like n_2 wakes up after the node n_1 has downloaded the new image completely and rebooted from the new image, n_2 will eventually hear the advertisement from n_1.

3.6 Conclusion

In this chapter, we presented a sensor network reprogramming protocol called Stream that significantly reduces the number of bytes to be transmitted over the wireless medium for reprogramming. It addresses a fundamental problem in all existing network reprogramming protocols, whereby the application image together with the reprogramming protocol image is transferred. Stream pre-installs the reprogramming protocol image in a node and transfers the application image with a small addition. Consequently, it reduces the reprogramming time, number of bytes transferred, and the energy expended. Stream is implemented in TinyOS for the Mica family of sensor nodes. Experiments are conducted on a testbed of Mica2 motes to demonstrate the efficiency of Stream over Deluge. Our experiments show that Deluge requires up to 98% more reprogramming time and transfers up to 132% more number of bytes compared to Stream. Simulation experiments are conducted using TOSSIM to show the scalability of Stream and increasing advantages over Deluge with larger network sizes. We also presented an opportunistic sleeping mode of operation for Stream that lets nodes in the network sleep till the new code image reaches the neighborhood of the node. This extension is analyzed, simulated, and empirically demonstrated to achieve energy savings, which become significant for large networks.

4. SINGLE VERSUS MULIT-HOP REPROGRAMMING

In recent years, the focus of the sensor network reprogramming has shifted from single hop reprogramming (only nodes within the transmission range of the base node (BN) are reprogrammed) to multi-hop reprogramming (all nodes in the multi-hop network are reprogrammed) because of various reasons. First and perhaps the biggest advantage is that from user's point of view, it is tedious to perform many rounds of single hop reprogramming to completely reprogram the multi-hop network. Second, multi-hop reprogramming protocols like Deluge [92], Freshet [102], and Stream [88] spatially pipeline the code transfer (also called spatial multiplexing) and thus reduce the time to reprogram the network. That is, a node does not need to completely download the code image before starting to send the code to its neighbors. These protocols divide the entire code image into pages consisting of fixed number of packets. When a node completes downloading a single page, it can send that page to other nodes in the network.

But in some deployment conditions, like in combined Sewage Overflow (CSO) [81] project implemented in South Bend, Indiana, multi-hop reprogramming can be costly in terms of reprogramming time and energy. In CSO, a multi-hop sensor network, called CSOnet, with nodes mounted on traffic lights and lamp posts, is used to collect alerts from monitoring sensors planted in the manholes of the municipal sewage system. The network then forwards these alerts to gateways at major traffic intersections which make distributed control decisions to channel the flow to temporary reservoirs so that dumping the waste water into rivers or lakes can be avoided.

At first glance, it may appear pointless to sacrifice the relative ease of the multi-hop reprogramming in favor of node by node reprogramming. The conditions in which a sensor network is deployed may change over time. For example, the link reliabilities between the nodes in the network may change because of varying environmental fac-

tors. When link reliabilities are low, sending entire application image over multiple links imposes a heavy burden in terms of retransmissions. This increases the reprogramming cost — both reprogramming energy and time — and congestion in the wireless links which may be better utilized in transferring critical data. In fact, for many current reprogramming protocols, except Stream, what needs to be transferred over the network is the entire application image plus the reprogramming protocol image. This exacerbates the problem by increasing the number of packets that needs to be transmitted reliably through the network.

This specific problem reared its head in the CSOnet deployment where it was observed that the batteries were being drained much faster than the theoretical calculations had predicted. Our investigation revealed that regular code updates being sent using the multi-hop method were the culprit for parts of the network, particularly the parts having linear topology and unreliable links. We decided to explore the possibility of judiciously using single hop reprogramming. In contrast to multi-hop reprogramming, in the single-hop method, the user visits each node in the deployment field and remotely reprograms it being physically as close as possible to the node. The severity of the above problem can thus be greatly reduced because the user goes as close as possible to the node to be reprogrammed to maximize link reliability. This reduces the number of retransmissions and hence reprogramming time and energy will be conserved. Generally hardwired reprogramming (by directly connecting the sensor node to the computer via say serial port) cannot be a substitute for single hop reprogramming to tackle the high cost of multi-hop reprogramming. For example, in the CSOnet deployment, since the sensor nodes are situated on top of the traffic posts, it is tedious and difficult to bring down the sensor nodes from the traffic posts and manually upload the code to these sensor nodes. The company responsible for the implementation of the project — EmNet LLC [81] in Granger, Indiana — reports high cost and logistical difficulties in reprogramming the sensors manually. This mode of operation cost EmNet $200 to reprogram each node including 3 persons involved and the rental cost of a bucket truck. Moving to a single hop reprogramming brings

the cost down by a factor of 10 and therefore, economically, the single hop wireless reprogramming appears a good compromise.

In this chapter, we present a protocol called DStream having both single and multi-hop reprogramming capabilities. We use the terms DStream-SHM and DStream-MHM to represent the single and multi-hop reprogramming modes of Stream. Using mathematical analysis, testbed experiments and simulations, we draw valuable inferences about the two reprogramming approaches. The common insight that all three gives us is that single hop may be more energy efficient and faster than multi-hop in some scenarios. For a given topology, the cutoff depends on the link reliability of the links in the network. High link reliability favors multi-hop reprogramming. However the cross-over point depends on which metric is of interest to the network owner — if it is reprogramming time, the cross-over happens at a lower link reliability value than for energy. Second, for networks that are linear (or close to linear), single hop reprogramming tends to be favored since a single broadcast of the code image can satisfy only a few nodes. The actual choice between the two modes will also be determined by the human cost of reprogramming a node at a time as in single hop reprogramming. For reference, we quantify this value for the CSOnet deployment.

To summarize, our contributions are:

1. Motivate the community to consider situations where single hop method may be more attractive than the currently held view of multi-hop reprogramming.

2. Design a dual reprogramming protocol, DStream, that does not significantly increase the code size or the memory footprint over the previous Stream protocol.

3. Through analytical, experimental and simulation results, provide a set of guidelines that help the network owner to choose single or multi-hop reprogramming approach based on current network conditions.

4.1 Protocol Design

4.1.1 Background and Rationale

It is desirable to have the sensor nodes equipped with the facility of both single and multi-hop reprogramming so that a choice can be made at runtime based on the current network conditions (topology, link reliabilities, density etc). The obvious approach is to have two separate reprogramming protocols (a single hop protocol like XNP [100] and a multi-hop protocol like Stream) stored in each nodes permanent storage (external flash) so that it can run the appropriate protocol when required by loading that protocol from external flash to the program memory. This is not an attractive solution because requiring a node to store two reprogramming protocols decreases the storage (e.g. external flash for Mica2 is 512KB) for the application running on the nodes. Our proposed approach is to have a single protocol with both single and multi-hop reprogramming capabilities. Existing single-hop reprogramming protocols, such as XNP, were not designed with the ability of propagating the code updates through the network in a multi-hop manner. Therefore they cannot serve as a starting point for our protocol. Multi-hop reprogramming protocols like Deluge, Stream and Freshet are more suited for this purpose. Since Stream is the most energy efficient and fastest among these protocols, we chose Stream and modified it to DStream, having both single and multi-hop reprogramming capabilities.

For this chapter, the meaning of single hop reprogramming is that only a single node, specified by the user, within single hop of the BN is reprogrammed. Contrary to what the name suggests, single hop reprogramming does not mean that all the nodes within the single hop of the BN are reprogrammed by this approach. This is because the main rationale behind single hop reprogramming is to avoid reprogramming nodes which have low link reliability to the BN but may technically be considered within a single hop of the BN. If we attempt to reprogram a node within single hop of the BN but with low link reliability, this may take considerable time and energy to be reprogrammed, defeating the purpose of single hop reprogramming.

4.1.2 Design Approach of DStream

Next we describe DStream that can provide both single and multi-hop reprogramming features. Let initially all nodes have Stream-RS as image 0 and the application with Stream-AS as image 1. Each node is executing the image 1 code. Consider that a new user application has to be injected into the network.

1. If multi-hop reprogramming is to be used, in response to the reboot command from the user, all nodes in the network reboot from image 0. This is accomplished as follows:

   (a) From the computer, the user sends the command to reboot from image 0 to the BN.

   (b) The BN executing image 1 broadcasts the reboot command to its one hop neighbors and itself reboots from image 0.

   (c) When a node running the user application receives the reboot command, it rebroadcasts the reboot command and reboots from image 0.

2. If single hop reprogramming is to be used, in response to the reboot command from the user, a single node specified by the user reboots from image 0. This is accomplished as follows:

   (a) From the host computer, the user sends the command to reboot a single node, say node α, from image 0 to the BN.

   (b) The BN running image 1 broadcasts the reboot command along with the user specified node id α to its one hop neighbors. The BN then reboots from image 0.

   (c) Each node that receives the reboot command, determines if the reboot command is targeted to it. If yes, it reboots from image 0. Otherwise, it ignores the reboot command. So, only the node α reboots from image 0 (Stream-RS) and is subsequently reprogrammed.

3. Stream-RS starts to reprogram the node(s) that has rebooted from image 0. Thus, Stream-RS which forms the bulk of the reprogramming protocol does not need any modification to support the single-hop mode of operation.

4. Stream-RS uses the three way handshake method for reprogramming [92] where each node broadcasts the advertisement about the code pages that it has. When a node hears the advertisement of newer data than it currently has, it sends a request to the node advertising newer data. Then the advertising node broadcasts the requested data. Each node maintains a set S containing the node ids of the nodes from which it has received the requests.

5. Once the node downloads the new user application completely, it performs a single-hop broadcast of an ACK indicating it has completed downloading. In single-hop reprogramming, only one node sends the ACK while in multi-hop all nodes in the network are ultimately reprogrammed and send the ACK message.

   When a node n_1 receives the ACK from node n_2, n_1 removes the id of n_2 from the set S. Note that in multi-hop reprogramming case, set S is maintained by all the nodes that are participating in sending code to any of its neighbors, while only the BN has a non-empty set S in single hop reprogramming and it only contains the node id α.

6. When the set S is empty and all the images are complete (by complete we mean that all pages of all images have been downloaded), the node reboots from image 1. So, in multi-hop case, at completion, the entire network is reprogrammed and all nodes reboot from image 1. In the single hop case, the set S is always empty for the node α that is reprogrammed and hence immediately after it completes downloading the image, the node α sends ACK and reboots from image 1. When the BN receives the ACK from the node α, it removes the id of node α from its set S and reboots from image 1.

From the above discussion, it is clear that DStream can provide both multi-hop and single hop reprogramming features. If the user specifies the id of the node to

be reprogrammed in the reboot command, DStream reprograms only the specified node (single hop reprogramming). Besides this, the user can also specify an option (switch_SH) for automatic switching between single and multi-hop approaches. When this option is specified, DStream starts with multi-hop reprogramming. When a node n_1 receives a request from a node n_2 for a page of the new image, n_1 keeps track of how many packets are requested for the same page in the next request by n_2. This gives n_1 the estimate of the link reliability between n_1 and n_2. If the estimated link reliability is less than some threshold (user specified), a message is sent back to the BN informing it about the current link reliability between n_1 and n_2. The BN then forwards that message to the computer. This suggests the user to switch to single hop reprogramming for n_2. In this way, nodes with low link qualities are reprogrammed using single hop method and other nodes are reprogrammed using multi-hop method.

4.2 Mathematical Analysis

Here we present an approximate analysis of the reprogramming time and energy for DStream-SHM and DStream-MHM for linear and grid networks. For linear networks, we assume that the spacing between consecutive nodes is equal to the transmission range and for grid networks, it is $\sqrt{2}$ times the grid spacing. Let the application consist of N_p pages with A_{pkt} packets per page. Let L_{RS} and L_{RM} be the link reliability of single hop reprogramming (for the link between the BN and the single node being reprogrammed) and multi-hop reprogramming (we assume identical link reliability for all links) respectively. Let P_s be the probability of successful transmission of a packet over a single link, which is equal to L_{RS} in single hop mode and L_{RM} in multi-hop mode.

4.2.1 Reprogramming Time

The reprogramming model and the assumptions that we use to compare the reprogramming times of single and multi hop modes of reprogramming are same as those in

the previous chapter (section 3.2). From equation 3.7, the multi hop reprogramming time is

$$T_{conv(M)} = T_r.E[N_{r,h_{max}}] \tag{4.1}$$

where

$$E[N_{r,h}] = min\left\{3.(N_P-1)+h, N_P.h\right\}.\sum_{i=1}^{\infty}\left[1-\left[\sum_{j=1}^{i-1}P_S(1-P_S)^{j-1}\right]^{N_{pkt}}\right]$$

For multi-hop reprogramming, $P_s = L_{RM}$. For single-hop reprogramming, $P_s = L_{RS}$, and the pages can not be pipelined. Therefore, the reprogramming time for the single-hop mode is

$$T_{conv(S)} = N.T_r.N_P\sum_{i=1}^{\infty}\left[1-\left[\sum_{j=1}^{i-1}L_{RS}(1-L_{RS})^{j-1}\right]^{A_{pkt}}\right] \tag{4.2}$$

The relative reprogramming time of single-hop to that of multi-hop is given by

$$\begin{aligned} T_{conv(S/M)} &= \frac{T_{conv(S)}}{T_{conv(M)}} \\ &= \frac{N.N_P\sum_{i=1}^{\infty}\left[1-\left[\sum_{j=1}^{i-1}L_{RS}(1-L_{RS})^{j-1}\right]^{A_{pkt}}\right]}{(3.(N_P-1)+h_{max})\sum_{i=1}^{\infty}\left[1-\left[\sum_{j=1}^{i-1}L_{RM}(1-L_{RM})^{j-1}\right]^{A_{pkt}}\right]} \end{aligned} \tag{4.3}$$

Using Equation 4.3, Figure 4.1 and Figure 4.2 show the relative reprogramming time (single hop/ multi-hop) respectively for linear and grid topologies as a function L_{RM} for different network sizes with L_{RS}=0.95, N_p=12 pages, A_{pkt}=48 packets, h_{max}=$N-1$, for the line topology, where N is the number of nodes, and $h_{max} = m-1$ for the nxm grid (ignoring the edge effects).

For the linear topology, as the network size increases the multi-hop mode reprogramming is faster due to the pipelining effect of multiple pages. However for the 5 node network, when the multi-hop link reliability is less than 0.8, single hop reprogramming is preferred from the delay point of view. For the grid topology, the reprogramming time of the multi-hop mode is always better than that of the single

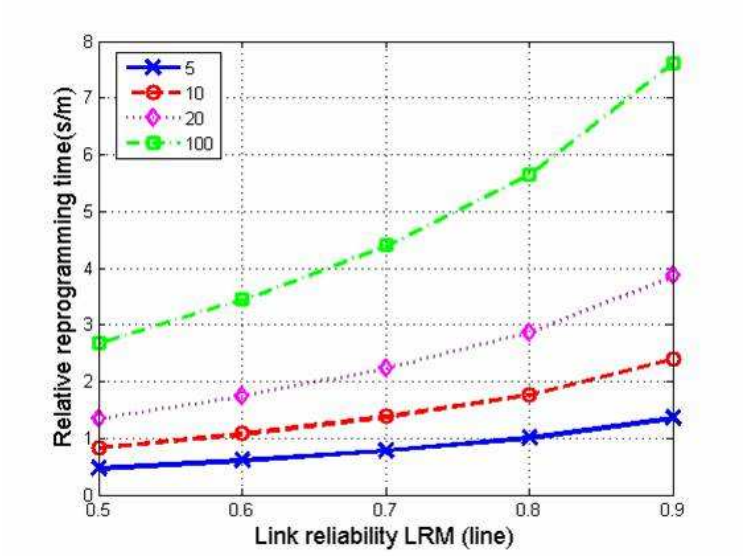


Fig. 4.1. Relative reprogramming time (single hop : multi-hop) as a function of link reliability for linear topologies

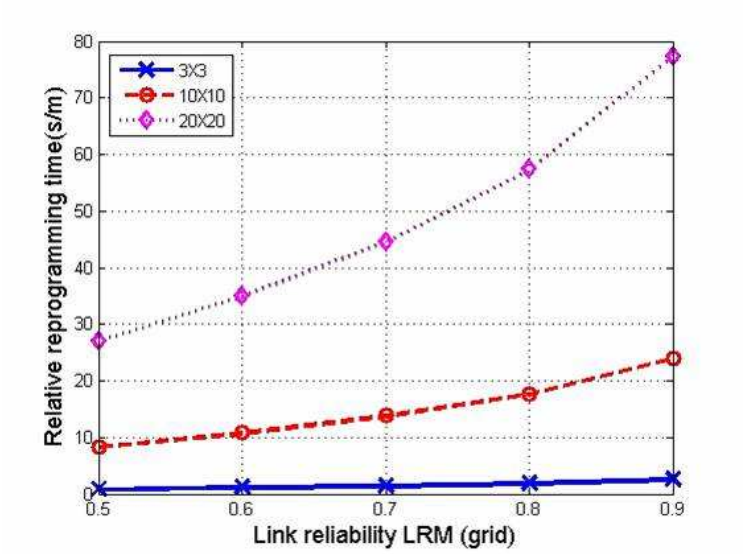


Fig. 4.2. Relative reprogramming time (single hop : multi-hop) as a function of link reliability for grid topologies

hop mode due to two factors — the spatial multiplexing and multiple nodes receiving the same single broadcast of the code packet. The spatial multiplexing becomes

more efficient with increasing network size, which explains the advantage of multi-hop reprogramming as network size increases.

4.2.2 Energy Cost

From equation 3.10 in the previous chapter, the total energy overhead of multi-hop reprogramming all the nodes in a network with maximum number of hops = h_{max} is

$$E_M = \sum_{h=1}^{h=h_{max}} E_h = \sum_{h=1}^{h=h_{max}} \left[\frac{N_P.N_{pkt}.C.\alpha_h}{L_{RM}^{|S_h|}} \right] \tag{4.4}$$

For a linear topology of N nodes with $R_{tx} = d$, where d the spacing between nodes, and R_{tx} is the transmission range, $\alpha_h = 1$, $|Sh| = 1$, and $h_{max} = (N-1)$. For an $n \times m$ grid topology, ignoring edge effects, with $r = \sqrt{2}d$, $\alpha_h = \lceil \frac{n}{2} \rceil$, $|Sh| = 3$, and $h_{max} = (m-1)$ (ignoring the edge effects). Let $N_{pkt} = A_{pkt} + 1 + E[N_r]$, where the second term is to account for the advertisement packet and the last term represents the expected number of request packets to successfully transmit the whole page. For single-hop reprogramming ($P_s = L_{RS}$), the total energy to reprogram all the nodes is given by

$$E_S = \frac{N_P.N_{pkt}.C.N}{L_{RS}} \tag{4.5}$$

The relative energy consumption of single-hop to multi-hop reprogramming is given by,

$$\begin{aligned} E_{S/M} &= \frac{E_S}{E_M} \\ &= \frac{N_P.\left(A_{pkt} + 1 + E_S[N_r]\right).C.N/L_{RS}}{\sum_{h=1}^{h=h_{max}} \left[\frac{N_P.\left(A_{pkt} + 1 + E_M[N_r]\right).C.\alpha_h}{L_{RM}^{|S_h|}} \right]} \\ &= \frac{N.L_{RM}^{|S_h|} \left(A_{pkt} + 1 + \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} L_{RS}(1-L_{RS})^{j-1}) \right]^{A_{pkt}} \right] \right)}{L_{RS} \sum_{h=1}^{h=h_{max}} (\alpha_h) \left(A_{pkt} + 1 + \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} L_{RM}^{|S_h|} \left(1 - L_{RM}^{|S_h|}\right)^{j-1} \right]^{A_{pkt}} \right] \right)} \end{aligned} \tag{4.6}$$

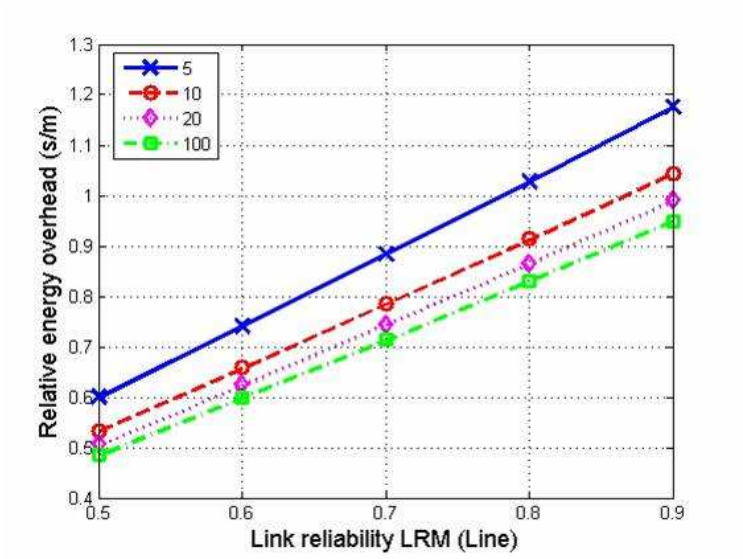


Fig. 4.3. Relative energy overhead (single hop : multi-hop) as a function of link reliability for linear topologies

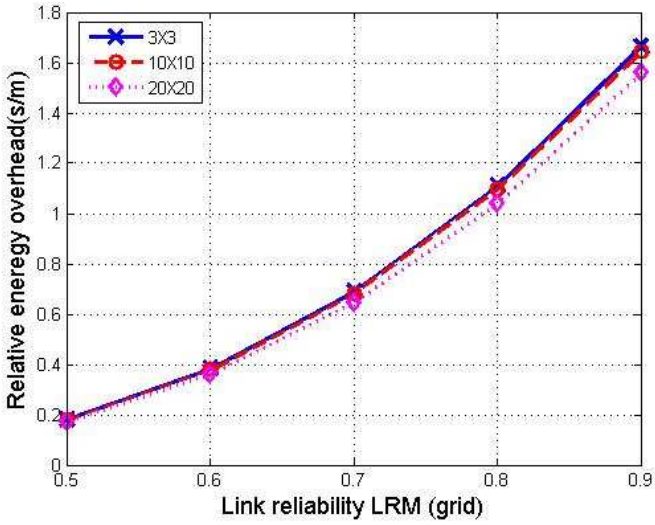


Fig. 4.4. Relative energy overhead (single hop : multi-hop) as a function of link reliability for grid topologies

Using equation 4.6, we plot relative energy overhead (single hop/ multi-hop) versus L_{RM} for linear and grid topologies for different network sizes. Figure 4.3 shows that the single hop mode is more efficient than the multi-hop mode for the linear topology

with link reliability less than 0.8. Moreover, the difference increases, in favor of the single hop mode, as the network size increases. In linear topologies, only one node can be satisfied by the transmission by a node and this negatively impacts the energy consumption of the multi-hop mode. This is due to the low link reliabilities with $|S_h| = 1$ for the line topology. Figure 4.4 shows that for a grid topology, almost irrespective of its size, the single hop mode is better when the link reliability is less than or equal to 0.8 and the multi-hop mode is better otherwise. Below multi-hop link reliability of 0.8, a redundant set of size $|S_h| = 3$ is not enough to compensate for the lower reliability, however, it becomes enough for multi-hop link reliabilities of more than 0.8. For a deployment with higher transmission ranges and hence higher values of $|S_h|$, the balance will shift in favor of multi-hop reprogramming.

4.3 Experiments and Results

We implement DStream using the nesC programming language in TinyOS. In this section, we compare the performances of DStream-SHM and DStream-MHM using both testbed experiments and simulations. The metrics that we use to compare single and multi-hop reprogramming approaches are reprogramming time and energy.

4.3.1 Calculation of Reprogramming Time and Energy

For multi-hop reprogramming, time to reprogram the network is the time interval between the instant t_0 when the BN sends the first advertisement packet to the instant t_1 when the last node (the one which takes the longest time to download the new application) completes downloading the new application. Since clocks maintained by the nodes in the network are not synchronized, we cannot take the difference between t_1 and t_0. Although a synchronization protocol can be used to solve this issue, we do not use it in our experiments because we do not want to add to the load in the network (due to synchronization messages) or the node (due to the synchronization protocol). Instead we follow the following approach. When the BN sends the first

advertisement packet, it reads its local clock and stores the current local time t_0^i in its external flash. Then it broadcasts a special packet called the sync packet after putting its node id i in the *src* field of the packet. It stores the time t_1^i when the sync packet is sent (i.e. when sendDone( ) event is signaled). Each node i in the network stores the local time t_0^i when it receives the first sync packet. It also stores the id of the node from which it received the first sync packet. Let us define a parent of a node i to be the node j from which the node i receives the first sync packet. Then the node i broadcasts the sync packet (with its id inserted into the src field) after random time uniformly distributed between some interval $(0, T)$. This is to avoid the collision of the sync messages broadcast by different nodes within the communication range of each other. Finally the node i stores the time t_2^i when it completes downloading all the pages of the new image. Note that a node i may receive many sync packets but it discards all of them except the first one. Also, a node sends a sync packet only once. So, this approach floods the sync packet across the network in a controlled manner. Let R_i be the reprogramming time for a node i — the time interval between the instant when the BN sends the first advertisement packet and the instant when the node i downloads the new code image completely. Let the parent of the node i be i_1 whose parent is i_2 and so on, and i_n is the BN. Reprogramming time R_i for node i is

$$R_i = (t_2^i - t_0^i) + \sum_{k=1}^{n} (t_1^{i_k} - t_0^{i_k})$$

Reprogramming time for the network is $max(R_i)$ over all nodes i in the network.

For DStream-SHM, we calculate the time t_s to reprogram a single node using the same method as explained above. Time to reprogram the network using single hop method is $R = N * t_s$ where N is the number of nodes in the network. Of course, we do not include the time required by the user to move from one node to another since such travel times differs from deployment to deployment. To compare the reprogramming times for single and multi-hop approaches for a given sensor network deployment, one should add these travel times to the single hop reprogramming times mentioned in

this thesis. Alternately, the reprogramming of the nodes can be done concurrently through multiple base stations at a higher resource cost.

Among the various factors that contribute to the energy used in the process of reprogramming, two important ones are the number of radio transmissions in the network and the number of flash-writes (the downloaded application is written to the external flash as image 1). Since the radio transmissions are the major sources of energy consumption and the number of writes to the external Flash is the same in the two cases (DStream-SHM and DStream-MHM), we take the total number of packets transmitted by all nodes in the network as the measure of energy used in reprogramming. The listening energy depends on two primary factors — the first is the time to complete reprogramming (which is already captured in our first metric) and the second is application policy about setting the node off to sleep (which is not related to the reprogramming protocol itself). The receiving energy and the listening energy are therefore neglected in the evaluation.

4.3.2 Testbed Description

We perform the experiments using Mica2 nodes having a 7.37 MHz, 8 bit microcontroller; 128KB of program memory; 4KB of RAM; 512KB external flash and 916 MHz radio transceiver. Testbed experiments are performed for three different network topologies: grid, linear and actual CSOnet networks (Figure 4.5). For each network topology, we define neighbors of a node n_1 as those nodes which can receive the packets sent by n_1. In our testbed experiments, if a node n_1 receives a packet from a node n_2 which is not its neighbor, the packet is dropped. Otherwise if n_1 and n_2 are neighbors, n_1 generates a random number u uniformly distributed in the interval [0,1] and if $u < L_{RM}$, then n_1 accepts the packet, otherwise the packet is dropped. This emulates different link reliabilities, since it is difficult to generate experimental conditions with exact link reliabilities. For the grid network used in our experiments, the transmission range R_{tx} of a node satisfies $\sqrt{2}d < R_{tx} < 2d$, where d

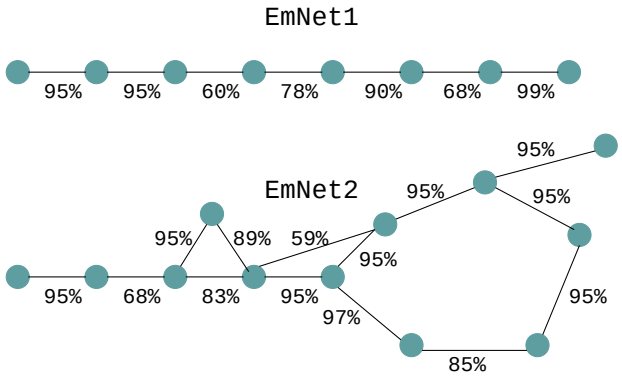


Fig. 4.5. Two CSOnet networks: EmNet1 and EmNet2

is the separation between the two adjacent nodes in any row or column of the grid. For the linear networks, $d < R_{tx} < 2d$. For multi-hop reprogramming of grid network, a node situated at one corner of the grid acts as the BN while the node at one end of the line is the BN for linear networks. For DStream-SHM, the link reliability of the single wireless link from the user to the one node being reprogrammed is kept constant (0.95) in the experiments. In practice, this is a high value since the user can get close to the node with the BN and there is no other transmission going on. In DStream-MHM, the link reliabilities L_{RM} of all links are identical and we vary it from 0.6 to 1.0 (perfect link). The link reliabilities shown in Figure 4.5 are derived from data collected over a summer period by doing a ping test with two radios with no other traffic in the CSOnet network. The values of link reliabilities among the nodes vary over different seasons of the year and even within the same season, the current environmental conditions may change these values from one day to another.

4.3.3 Testbed Experiment Results

Figure 4.6-a and Figure 4.6-b compare the average reprogramming time and energy for 2X2, 3X3 and 4X4 grid networks using DStream-SHM and DStream-MHM with different values of link reliabilities. These figures show that multi-hop reprogramming takes more time and energy to reprogram the network if link reliability

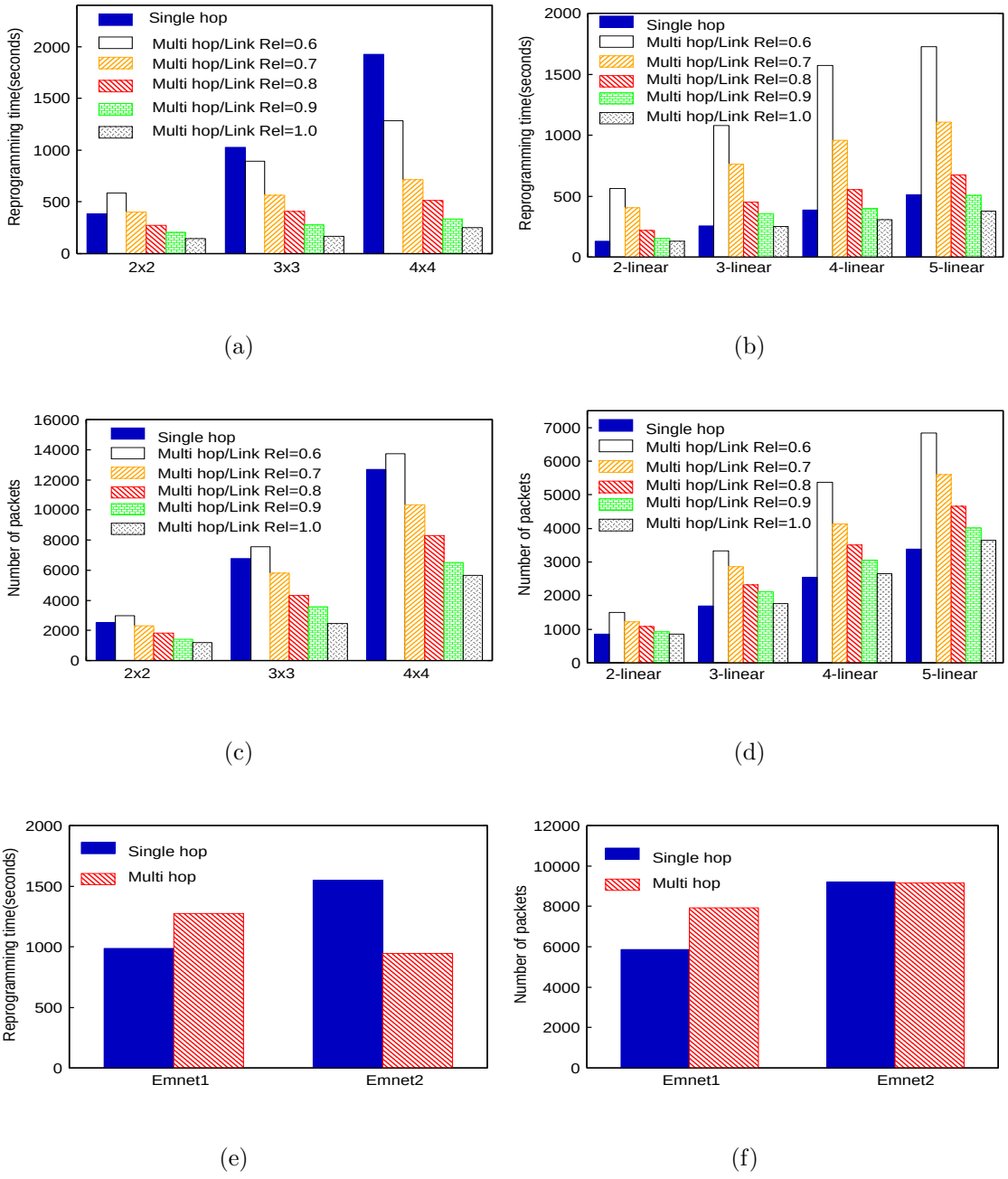


Fig. 4.6. Testbed results: Reprogramming time for (a) grid, (b) linear , and (e) CSOnet networks. Number of packets transmitted in the network during reprogramming for (c) grid, (d) linear, and (f) CSONet networks. For grid and linear topologies, the leftmost bar is reprogramming time for single hop and the remaining bars are multi-hop reprogramming times with increasing link reliabilities. The order of the legends is the order of the bars from left to right.

is decreased because of more retransmissions (and hence more time) required for a packet to be successfully received by the sensor node. Figure 4.6-a shows that in small networks (2X2 in the experiment), for $L_{RM} < 0.8$, single hop reprogramming is faster than multi-hop reprogramming. However, for larger networks, DStream-MHM is always better for the range of L_{RM} (0.6-1.0) considered in these experiments. But it should be noted that even in large grids, if we carry out the experiments for link reliabilities less than 0.6, then below some value r_t, single hop becomes faster than multi-hop reprogramming. Figure 4.6–b shows that there exists some value of link reliability $L_{RM} > 0.6$ for which multi-hop reprogramming takes less energy than single hop reprogramming. For good link reliabilities, multi-hop approach is faster and more energy efficient than single hop because of the following reasons: (1) Multiple listening nodes: In multi-hop reprogramming, a single broadcast of the data packet by a node can be received by all its neighbors simultaneously. On the other hand, in single hop reprogramming, a single broadcast of the data packet is received by only one node at a time. (2) Spatial multiplexing: In multi-hop reprogramming, spatial multiplexing of the code transfer makes reprogramming faster. Note that spatial multiplexing contributes in reducing the reprogramming time, not the energy. As link reliability decreases, the difference between single and multi-hop approaches in terms of both reprogramming time and energy decreases and for $r < r_t$, single hop reprogramming becomes faster and for $r < r_e$ single hop reprogramming is more energy efficient. An experimental observation is that $r_t \neq r_e$ in general; thus system designers have to make a decision depending on which metric is more important, energy or delay. In linear networks, the only advantage that multi-hop reprogramming has over single hop reprogramming is spatial multiplexing of the code transfer. By definition, a single broadcast cannot satisfy more than one node in linear networks and thus this factor cannot provide an advantage to DStream-MHM. Hence as shown in Figure 4.6-c and Figure 4.6-d, the advantage of DStream-MHM over DStream-SHM is not as pronounced as in grid networks. Further, spatial multiplexing helps to make reprogramming faster but does not contribute in reducing the reprogramming energy. As

a result, as shown in Figure 4.6-d single hop reprogramming is always more energy efficient than multi-hop reprogramming for linear networks. Since spatial multiplexing of the code transfer is effective for larger networks, multi-hop reprogramming incurs less delay than single hop reprogramming for large networks (for example in Figure 4.6-c, for networks having at least 4 nodes) for good link reliabilities.

We can conclude that for linear networks (or networks which are approximately linear, i.e. most of the nodes have degree 2) single hop reprogramming is always more energy efficient than multi-hop reprogramming and except for very high link reliabilities among the nodes, single hop method is also faster than multi-hop method. On the other hand, multi-hop reprogramming is faster and more energy efficient for reasonable link reliabilities in grid networks, with the advantage increasing with network size. However consider that for practical deployments other factors, such as travel times may be added to the cost of DStream-SHM.

Figure 4.6-e and Figure 4.6-f compare reprogramming time and energy for the two CSOnet networks (Figure 4.5). Since EmNet1 is a linear network, reprogramming energy for EmNet1 is always less for single hop case than the multi-hop case. Reprogramming time of EmNet1 is also less for single hop reprogramming than multi-hop reprogramming because some link reliabilities are very low (like 60% and 68%). Even though multi-hop reprogramming for EmNet1 has the advantage of spatial multiplexing of the code transfer which helps to reduce the reprogramming time, the disadvantage due to low link reliabilities outweighs this advantage. For EmNet2, multi-hop reprogramming is faster than single hop reprogramming because multiple listening nodes can receive the single broadcast of the data packet simultaneously and spatial multiplexing of the code transfer make multi-hop reprogramming faster. The reprogramming energy for single and multi-hop reprogramming are almost equal for EmNet2.

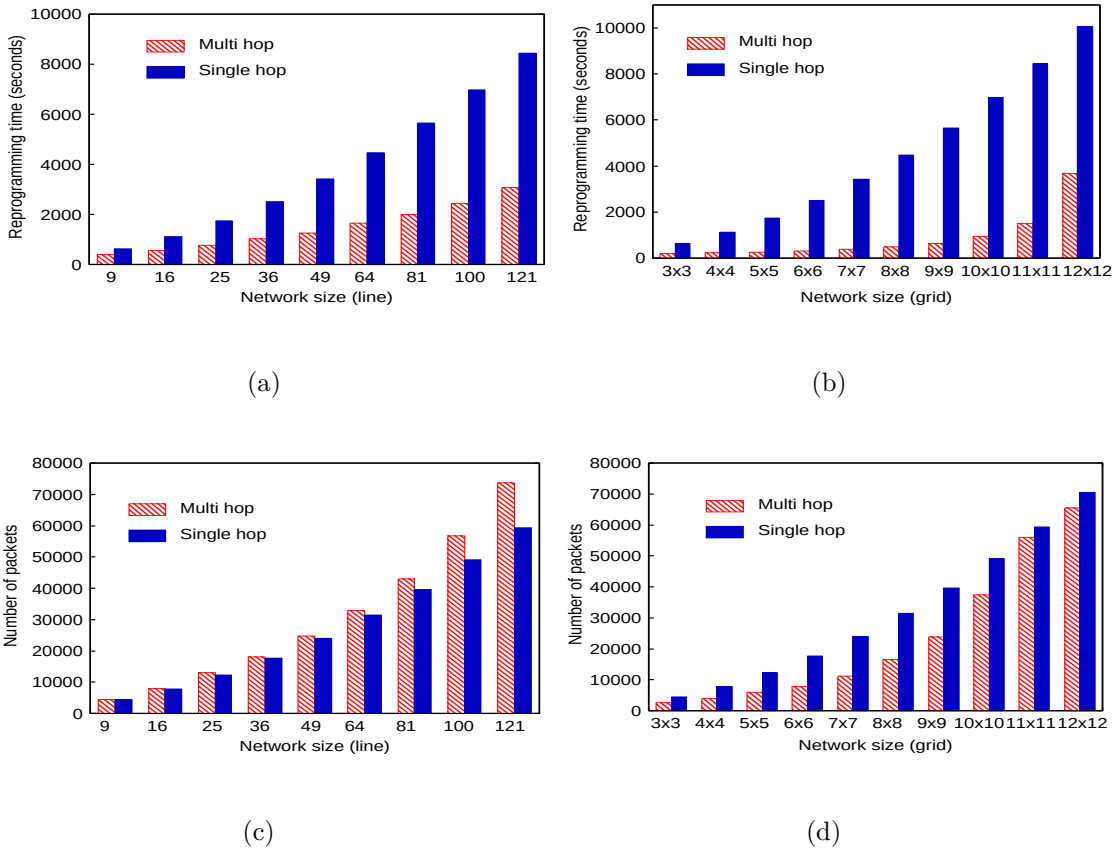


Fig. 4.7. Simulation results: Reprogramming time as a function of network size for (a) linear and (b) grid networks ($L_{RM} = 0.9$). Number of transmitted packets as a function of network size for (c) linear and (d) grid networks ($L_{RM} = 0.9$).

4.3.4 Simulation Results

We used TOSSIM simulator to examine the trend of overhead energy and reprogramming time for larger sized networks. We perform simulations for three different network topologies: grid, linear and random. The random topology is generated by uniformly distributing nodes with some given density over a square field. Figure 4.7-a to Figure 4.7-d compare DStream-SHM and DStream-MHM for linear and grid networks with $L_{RM} = 0.9$ and $L_{RS} = 0.95$. These results confirm with the analytical and testbed results.

Figure 4.8-a and Figure 4.8-b show the reprogramming time and the overhead energy respectively as a function of network density (shown as number of neighbors per node) for a random topology consisting of 100 nodes with $L_{RM} = 0.9$ and $L_{RS} = 0.95$. The figures show that the performance of multi-hop reprogramming improves as the network density increases. This is due to the increase in the number of nodes that can listen to the single broadcast of the code packet as the network density increases. Figure 4.8-c and Figure 4.8-d show the reprogramming time and the overhead energy respectively as a function of the multi-hop link reliability for a random topology with $N = 100$ and L_{RS}=0.95. Figure 4.8-c shows that multi-hop reprogramming is always faster and gets better as the multi-hop link reliability increases-again due to the pipelining of the code in multi-hop reprogramming. Figure 4.8-d shows that overhead energy of single hop reprogramming is lower than that of multi-hop reprogramming when the link reliability is less than or equal to 0.7 and the multi-hop mode is better otherwise. Below a link reliability of 0.7, the number of the nodes that can simultaneously receive the single broadcast of the code packet is not enough to compensate for the lower reliability. However, it becomes enough for link reliabilities of greater than 0.7. For a deployment with higher transmission ranges, the balance will shift in favor of multi-hop reprogramming.

4.4 Conclusion

Contrary to the prevalent idea explored in wireless reprogramming protocols, this chapter posits that single hop reprogramming can be a better choice under specific network conditions. To identify the conditions which favor single hop reprogramming, we performed mathematical analysis, testbed experiments (including experiments on real- world sensor networks) and simulations. Using Equation 4.3 and Equation 4.6, we can approximately find under what values of link reliabilities, and redundancy in the network, single hop can be better than multi-hop method in terms of reprogramming time and/or energy. Further from our mathematical analysis, testbed

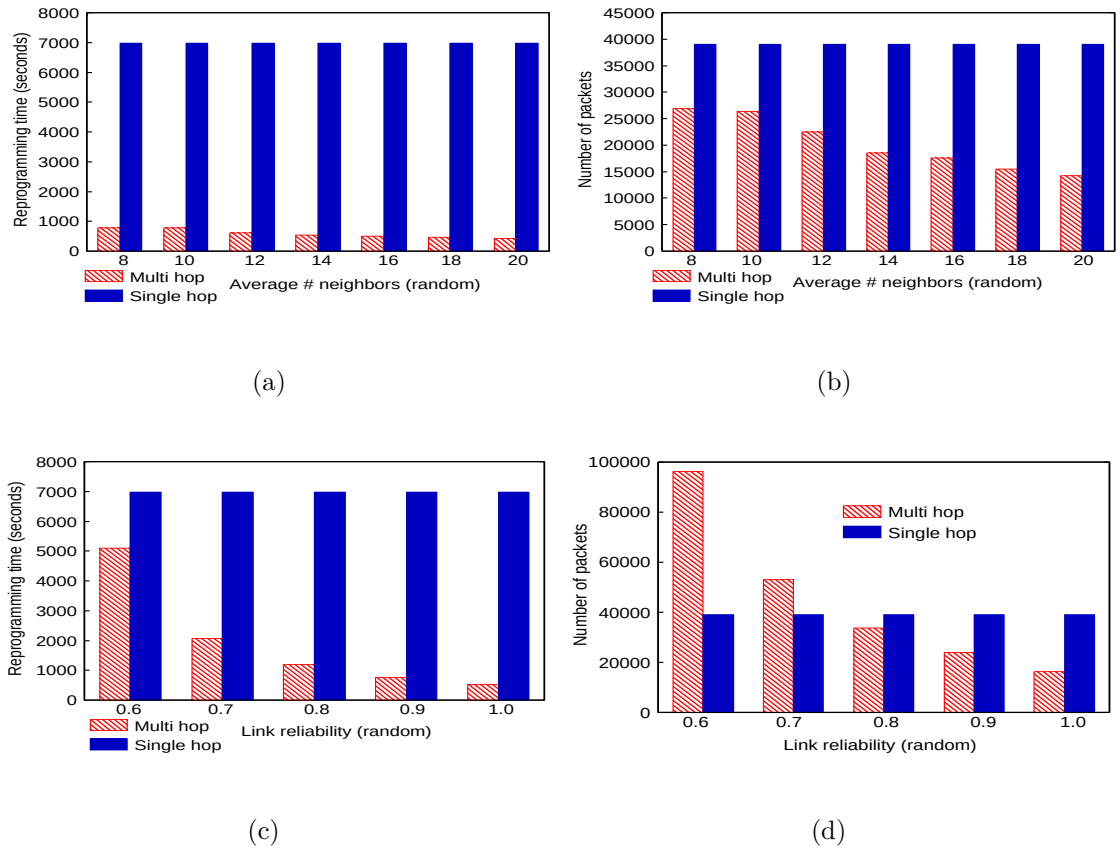


Fig. 4.8. Simulation results: (a) Reprogramming time and (b) number of transmitted packets as a function of network density (LRM=0.9) for random network topology; (c) Reprogramming time and (d) number of transmitted packets as a function of link reliability for 100-random topology (Mean number of neighbors=8).

experiments and simulations, we can provide the following insights which can serve as a guideline to the network-owner:

1. If the network is linear or approximately linear, single hop reprogramming is favored in terms of energy.

2. For smaller linear networks, single hop is faster than multi-hop if link reliabilities are poor. Our testbed results show that for a linear network consisting of 5 nodes, single hop is faster if link reliability is less than 0.9. Even for larger net-

works, if some of the links are very unreliable (as in the CSOnet deployments), single hop can be faster than multi- hop reprogramming. However as the network size increases, multi-hop improves relative to single hop since pipelining becomes more efficient.

3. For non linear networks, unless the link reliabilities are very poor, multi-hop reprogramming is both more energy efficient and faster than single hop. But single hop is worth considering if some links are really unreliable.

4. The exact cross-over link reliability below which single hop outperforms multi-hop depends on what metric we are interested in. If it is reprogramming time, then the cross-over value is lower than that for reprogramming energy.

5. With increasing density, multi-hop performs better since more number of nodes can be satisfied by a single broadcast of the code image. Also, this reaffirms the claim of Stream and Deluge that they are able to handle high network densities by appropriate collision arbitration schemes.

5. ZEPHYR: INCREMENTAL REPROGRAMMING USING FUNCTION CALL INDIRECTIONS

This chapter presents an incremental reprogramming system. The basic idea behind incremental reprogramming is that in practice, software running on a node evolves, with incremental changes to functionality, or modification of the parameters that control current functionality. The change may involve addition, deletion, or modification of one or a few components. Thus the difference between the currently executing code and the new code is often much smaller than the entire code. This makes incremental reprogramming attractive because only the changes to the code need to be transmitted, and the new application can be reassembled at the node from the existing application and the received changes. The goal of incremental reprogramming is to transfer a small delta, the difference between the old and the new software, so that reprogramming time and energy are minimized.

The design of incremental reprogramming protocol for sensor nodes poses several challenges. Many operating systems do not support dynamic linking of software components on a sensor node. For example, the standard release of TinyOS [14], one of the widely used operating systems for sensor nodes, does not provide this feature. TinyOS executes on each node a single monolithic image, that contains what would be called the kernel parts in a traditional OS, as well as the application parts. This rules out the straightforward transfer of only the components that have changed and dynamically linking them to at the node. The second class of operating systems, represented by SOS [10] and Contiki [15], do support dynamic linking. However, their reprogramming support also does not handle changes to the kernel modules. Moreover, the specifics of the position independent code strategy employed in SOS limit the kinds of changes to a module that can be handled. In Contiki, the requirement

to transfer the symbol and relocation tables to the node for runtime linking increases the amount of traffic that needs to be disseminated through the network.

In this chapter, we present a fully functional incremental multi-hop reprogramming protocol called *Zephyr*. It transfers the changes to the code, does not need dynamic linking on the node and does not transfer symbol and relocation tables. Dynamic linking on the nodes can violate the low overhead requirement of the sensor nodes because linker requires considerable computation resources and memory. Also, wireless transmission of symbol, relocation tables and other data structures for dynamic linking can consume significant energy. Zephyr uses an optimized version of the Rsync algorithm [93] to perform *byte-level comparison* between the old and the new code binaries. As we will show, even an optimized difference computation at the low level generates large deltas because of changes in the position of application components. Therefore, before performing byte-level comparison, Zephyr performs *application-level modifications*, the most important of which is to use function call indirections to mitigate the effects of changes in the location of functions caused by software modification. This requires a level of indirection for function calls but significantly reduces the size of delta where applications have high function reuse.

We implement Zephyr on TinyOS and demonstrate it using real multi-hop networks of Mica2 [1] nodes and through simulations. Zephyr can also be used with SOS or Contiki to upload incremental changes within a module, i.e. instead of transferring the entire module, only the difference between the old and the new modules can be transmitted to the sensor nodes. We evaluate Zephyr for a wide range of software change cases — from a small parameter change to almost complete application rewrite — using applications from both the TinyOS distribution and various versions of a real world sensor network application called eStadium [130] that has been deployed at the Ross-Ade football stadium at Purdue University. Our experiments show that Deluge [92], Stream [88], and the incremental protocol by Jeong and Culler [121] need to transfer up to 1987, 1324, and 49 times more number of bytes than Zephyr, respectively. This translates to a proportional reduction in reprogramming time and energy

for Zephyr. Furthermore, Zephyr enhances the robustness of the reprogramming process in the presence of failing nodes and lossy or intermittent radio links typical in sensor network deployments because of the significantly smaller amount of data that it needs to transfer across the network. Zephyr suffers from fewer link failures and retransmissions than the existing protocols because the amount of data that needs to be transmitted across the network is significantly less in Zephyr.

Our contributions in this chapter are as follows:

1. We present a technique that uses optimized byte-level comparisons and leads to small deltas.

2. We present application-level modifications that increase the structural similarity between different software versions, also leading to small delta.

3. We present techniques that support modification of any part of the software (i.e. kernel and user code), without requiring dynamic linking on sensor nodes.

4. We present the design, implementation and demonstration of a fully functional multi-hop reprogramming system. Most previous works have concentrated on some of the stages of the incremental reprogramming system, but have not delivered a functional complete system.

5.1 High level overview of Zephyr

Figure 5.1 is the schematic diagram showing various stages of Zephyr. First Zephyr performs application-level modifications on the old and new versions of the software to mitigate the effect of shifts in the function locations (hereafter called function shifts) so that the similarity between the two versions of the software is increased. Next the two executables are compared at the byte-level using a novel algorithm derived from the Rsync algorithm [93]. This produces the delta script which describes the differences between the old and new versions of the software. These computations are performed on the host computer. The delta script is then transmitted wirelessly

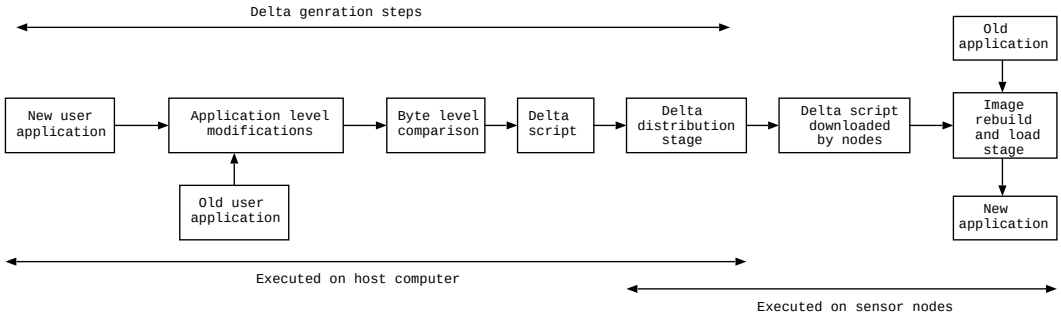


Fig. 5.1. Overview of Zephyr

to all the nodes in the network in the delta distribution stage. In this stage, first the delta script is injected by the host computer to the base node (a node physically attached to the host computer via, say, a serial port). The base node then wirelessly sends the delta script to all nodes in the network, in a multi hop manner, if required. The nodes save the delta script in their external flash memory. After the sensor nodes download the delta script, they build the new image using the delta and the old image and store it in the external flash. Finally the bootloader loads the newly built image from the external flash to the program memory and the node runs the new software.

We describe these stages in the following sections. We first describe byte-level comparison and show why it is not sufficient and thus motivate the need for application-level modifications.

5.2 Byte-level comparison

We first describe the Rsync algorithm [93] and then our extensions to reduce the size of the delta script that needs to be disseminated.

5.2.1 Application of Rsync algorithm

The Rsync algorithm was originally developed to update binary data between computers over a low bandwidth network. Rsync divides the files containing the bi-

nary data into fixed size blocks and both the sender and the receiver compute the pair (Checksum, MD4) over each block. If this algorithm is used as is for incremental reprogramming, then the sensor nodes need to perform an expensive MD4 computation for each block of the binary image. We modify Rsync such that all the expensive operations regarding delta script generation are performed on the host computer and not on the sensor nodes. The modified algorithm runs on the host computer only and works as follows: 1) The algorithm first generates the pair (Checksum, MD4 hash) for each block of the old image and stores it in a hash table whose key is the checksum. 2) The checksum is calculated for the first block of the new image. 3) The algorithm checks if this checksum matches the checksum for any block in the old image using a hash-table lookup. If a matching block is found, Rsync checks if their MD4 hashes also match. If MD4 hashes also match, then that block is considered as a matching block. Note that if two blocks do not have the same checksum, then MD4 is not computed for that block. This ensures that the expensive MD4 computation is done only when the inexpensive checksum matches between the two blocks. If no matching block is found then the algorithm moves to the next byte in the new image and the same process is repeated until a matching block is found. While the probability of collision is not negligible for two blocks having the same checksum, but with MD4 the collision probability *is* negligible. To ensure the correctness of our scheme in the rare case when two different blocks have the same MD4 hash, the algorithm can be modified to perform byte-by-byte comparison when MD4 hashes match. Since this algorithm runs on a powerful host computer, this is not a problem.

After running this algorithm, Zephyr generates a list of COPY and INSERT commands for matching blocks and non matching portions respectively (the size of the non-matching portions may not be equal to the block size):

```
COPY <oldOffset> <newOffset> <len>
INSERT <newOffset> <len> <data>
```

The COPY command copies *len* number of bytes from *oldOffset* at the old image to *newOffset* at the new image. Note that *len* is equal to the block size used in the

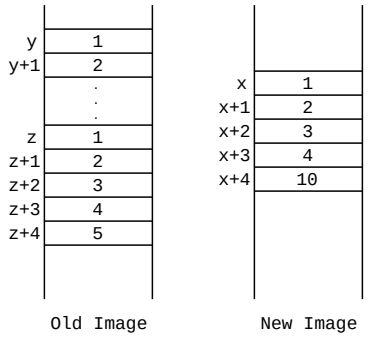


Fig. 5.2. Finding super block

Rsync algorithm. The INSERT command inserts *len* number of bytes, i.e. *data*, to *newOffset* of the new image. Note that this *len* is not necessarily equal to the block size or its multiple.

5.2.2 Rsync optimization

With the Rsync algorithm, if there are n contiguous blocks in the new image that match n contiguous blocks in the old image, n COPY commands are generated. We change the algorithm so that it finds the largest contiguous matching block between the two binary images. Note that this does not simply mean merging n COPY commands into one COPY command. As shown in Figure 5.2, let the blocks at the offsets x and $x+1$ in the new image match those at the offsets y and $y+1$ respectively in the old image. Let blocks at x through $x+3$ of the new image match those at z through $z+3$ respectively of the old image. Note that blocks at x and $x+1$ match those at y and $y+1$ and also at z and $z+1$. The Rsync algorithm creates two COPY commands as follows: COPY $\langle y \rangle$ $\langle x \rangle$ $\langle B \rangle$ and COPY $\langle y{+}1 \rangle$ $\langle x{+}1 \rangle$ $\langle B \rangle$, where B is the block size. Simply combining these 2 commands as COPY $\langle y \rangle$ $\langle x \rangle$ $\langle 2{*}B \rangle$ does not result in the largest contiguous matching block. The blocks at the offsets z through $z+3$ form the largest contiguous matching block. We call contiguous matching blocks a *super-block* and the largest super-block the *maximal super-block*.

The optimized Rsync algorithm finds the maximal super-block and uses that as the operand in the COPY command. Thus, optimized Rsync produces the single COPY command as COPY $\langle z \rangle$ $\langle x \rangle$ $\langle 4*B \rangle$ for the example just given. Figure 5.3 shows the pseudo code for optimized Rsync. Its complexity is $O(n^2)$ where n is the number of bytes in the image. This is not a concern because the algorithm is run on the host computer and not on the sensor nodes, and is run only when a new version of the software needs to be disseminated. As we will show in Section 8.2, optimized Rsync running on the desktop computer took less than 4.5 seconds for a wide range of software change cases that we experimented with.

5.2.3 Drawback of using only byte-level comparison

To see the drawback of using optimized Rsync alone, we consider two cases of software changes.

Case 1: Changing Blink application: Blink is an application in the TinyOS distribution that blinks an LED on the sensor node every second. We change the application from blinking a green LED every second to blinking it every two seconds. Thus, this is an example of a small parameter change. The delta script produced with optimized Rsync is 23 bytes long which is small and congruent with the actual amount of change made in the software.

Case 2: We added four lines of code to Blink. The delta script between the Blink application and the one with these four lines added is 2183 bytes. The change made in the software for this case is slightly more than that in the previous case, but the delta script produced by optimized Rsync in this case is disproportionately larger.

When a single parameter is changed in the application, as in Case 1, no part of the already matching binary code is shifted. All functions start at the same location as in the old image. But with the few lines added to the code (as in Case 2), the functions following the added lines are shifted. As a result, all the calls to those functions refer

```
/* Terminology
mbl=matching block list
cbl=contiguous block list
*/
1.   j=0 and cblStretch=0
2.   while j< number of bytes in the new image
3.     mbl=findAllMatchingBlocks(j)
4.     if mbl is empty
5.         j++
6.         if cbl is not empty
7.             Store any one element in cbl as maximum superblock
8.     else
9.        j=j+blockSize
10.        if (cblStretch==0)
11.          cbl=mbl
12.          cblStretch++
13.        else
14.          Empty tempCbl
15.          for each element in cbl do
16.            if (cbl.element + cblStretch == any entry in mbl )
17.              tempCbl=tempCbl U {cbl.element}
18.          if tempCbl is empty
19.            Store any one element in cbl as maximum superblock
20.            Empty cbl
21.            cblStretch=0
22.          else
23.            cbl=tempCbl
24.            cblStretch++
25. end while
findAllMatchingBlocks ( j )
     /*Same as Rsync algorithm, but instead of  returning the offset
     of just one matching block, returns a linked list consisting
     of offsets of all matching blocks in the old image for the
     block starting at offset j in the new image.*/
```

Fig. 5.3. Pseudo code of optimized Rsync that finds maximal super block

to new locations. This produces several additional changes in the binary file resulting in the large delta script.

The boundaries between blocks can be defined by Rabin fingerprints as is done in [131,132]. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. These fingerprints are efficient to compute on a sliding window in a file. It should be noted that a Rabin fingerprint can be a substitute for a byte-level comparison only. Because of the content-based boundary between the chunks in Rabin fingerprint approach, the editing operations change only the chunks affected by those edits even if they change the offsets. Only the chunks that have changed need to be sent. But when the function addresses change, all the chunks containing calls to those functions change, and need to be sent explicitly. This results in a large delta—comparable to the delta produced by the optimized Rsync algorithm without application-level modifications. Also the *anchors* that define the boundary between the blocks have to be sent explicitly. The chunks in Rabin fingerprints are typically quite large (8 KB compared to less than 20 bytes for our case). As we can see from Figure 5.7, the size of the difference script will be much larger at 8 KB than at 20 bytes.

5.3 Application-level modifications

The size of the delta script produced by a byte-level comparison is not always consistent with the extent of the change made in the software. This is a direct consequence of neglecting the structure of the code at the application level of the software and using only the binary comparison for generating the delta script. So we need to make modifications at the application level so that the subsequent stage of byte-level comparison produces delta script that is congruent in size with the amount of software change. One way of tackling this problem is to leave some slop (empty) space after each function as in [122]. With this approach, even though a function expands (or shrinks), the location of the following functions will not change as long as

the expansion is accommodated by the slop region assigned to that function. But this approach wastes program memory, and thus is not desirable for memory-constrained sensor nodes. Also, this approach creates a host of complex management issues such as what should be the size of the slop region (possibly different for different functions), and what should be done with the empty memory space caused by relocation of functions when they expand beyond the assigned slop region. Choosing too large of a slop region means wasting too much memory and too small a slop region means functions frequently need to be relocated, leading to large differences in the binary images. Another way of mitigating the effect of function shifts is by making the code position independent [10]. Position independent code (PIC) uses relative jumps instead of absolute jumps. However, not all architectures and compilers support this. For example, the AVR platform allows relative jumps within 4KB only and for MSP430(used in Telos nodes), no compiler is known to fully support PIC.

5.3.1 Function call indirections

For the byte-level comparison to produce a small delta script, it is necessary to make structural adjustments at the application-level to preserve maximum similarity between the two versions of the software. For example, let the application shown in Figure 5.4-a be changed such that the functions fun_1, fun_2, and fun_n are shifted from their original positions *b, c*, and *a* to new positions b', c', and a' respectively. Note that there can be (and generally will be) more than one call to a function. When these two images are compared at the byte-level, the delta script will be large because all the calls to these functions in the new image will have different target addresses from those in the old image. The approach we take to mitigate the effects of function shifts is as follows: Let the application be as shown in Figure 5.4-a. We modify the linking stage of the executable generation process to produce the code as shown in Figure 5.4-b. Here calls to functions $fun_1, fun_2, ..., fun_n$ are replaced by jumps to *fixed locations* $loc_1, loc_2, ..., loc_n$ respectively. In common embedded platforms, the

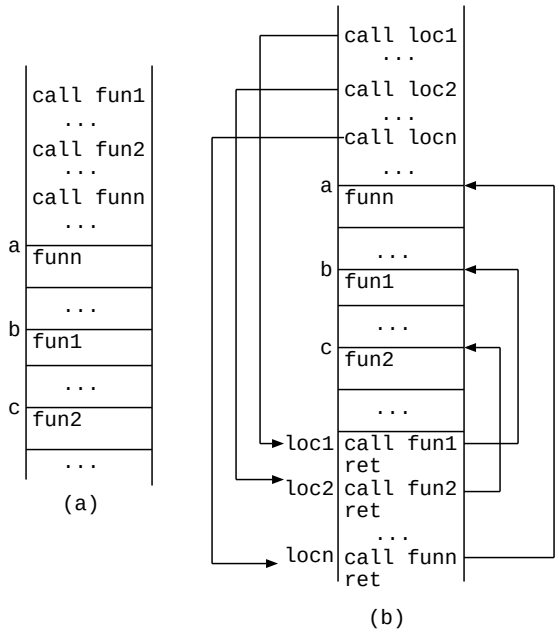


Fig. 5.4. Program image (a) without indirection table and (b) with indirection table.

call can be to an arbitrarily far off location. The segment of the program memory starting at the fixed location loc_1 acts like an indirection table. In this table, the actual calls to the functions are made. When the call to the actual function returns, the indirection table directs the flow of the control back to the line following the call to $loc_x (x = 1, ..., n)$. The location of the indirection table is kept fixed in the old and the new versions to reduce the size of the delta.

When the application shown in Figure 5.4-a is changed to the one where the functions $fun_1, fun_2, ..., fun_n$ are shifted, during the process of building the executable for the new image, we add the following features to the linking stage: When a call to a function is encountered, the linker checks if the indirection table in the old file contains the entry for that function (we also supply the old file (Figure 5.4-b) as an input to the executable generation process). If it does, then it creates an entry for that function in the indirection table in the new file at the same location as in the old file. Otherwise it assigns a slot in the indirection table for the function (call it a

rootless function) but does not yet create the slot. After assigning slots to the existing functions, it checks if there are any empty slots in the indirection table. These would correspond to functions which were called in the old file but are not in the new file. If there are empty slots, it assigns those slots to the rootless functions. If there are still some rootless functions without a slot, then the indirection table is expanded with new entries to accommodate these rootless function. Thus, the indirection table entries are naturally garbage collected and the table expands on an as-needed basis. As a result, if the user program has n calls to a particular function, they refer to the same location in the indirection table and only one call, namely the call in the indirection table, differs between the two versions. On the other hand, if no indirection table were used, all the n calls would refer to different locations in the new application than in the old one.

This approach ensures that the segments of the code, except the indirection table, preserve the maximum similarity between the old and new images because the calls to the functions are redirected to the fixed locations even when the functions have moved in the code. The basic idea behind function call indirections is that the location of the indirection table is fixed and hence the target addresses of the jump to the table are identical in the old and new versions of the software. If we do not fix the location of the indirection table, the jump to the indirection table will have different target addresses in the two versions of the software. As a result, the delta script will be large. In situations where the functions do not shift (as in Case 1 discussed in Section 5.2.3) Zephyr will not produce a delta script larger than optimized Rsync does without an indirection table. This is due to the fact that the indirection tables in the old and the new software match and hence Zephyr finds the large super-block that also contains the indirection table.

The linking changes in Zephyr are transparent to the user. She does not need to change the way she programs. The linking stage automatically makes the above modifications. We use linker command language to implement function call indirec-

tions. Zephyr does introduce one level of indirection during function calls (e.g., 8 clock cycles on the AVR platform).

5.3.2 Pinning the interrupt service routines

It should be noted that changes in the application software can cause changes not only in the positions of the user functions but also the positions of interrupt service routines. Such routines are not explicitly called by the user application. In most microcontrollers, there is an interrupt vector table at the beginning of the program memory, typically after the reset vector at 0x0000. Whenever an interrupt occurs, the control goes to the appropriate entry in the vector table that causes a jump to the required interrupt service routine. Zephyr does *not* change the interrupt vector table to direct the calls to the indirection table (as described above for the normal functions). Instead it modifies the linking stage to always put the interrupt service routines at fixed locations in the program memory so that the targets of the calls in the interrupt vector table do not change. This further preserves the similarity between the versions of the software. This approach is based on the assumption that interrupt service routines generally do not change. If interrupt service routines change, it does not cause a correctness problem, but causes the delta script to be larger.

5.3.3 Handling function pointers

Other than function call statements, shift in function addresses resulting from software changes can affect statements that use the function address. One important example of this is any instruction that uses function pointers. When a function is shifted in memory due to changes in the software, the instructions that use a function pointer referencing the shifted function also change between the old and the new versions of the software, causing the delta script to be large. Function pointers can be used in different ways by different systems. In this section, we take the function

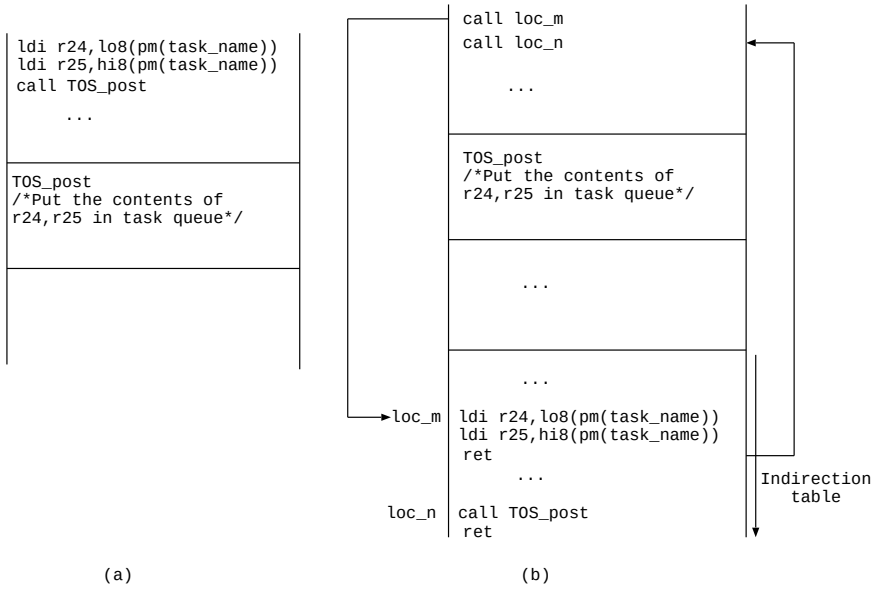


Fig. 5.5. Program image (a) without and (b) with handling problem due to function pointer.

pointer usage in TinyOS as an example to illustrate how the concept of indirection can be extended to handle such situations.

Function pointers are used in TinyOS to put *tasks* in the *task queue*. TinyOS is an event-based operating system which uses two computational abstractions—asynchronous events and synchronous tasks. An event-based application is implemented as a set of event-handlers and tasks. Tasks are a form of deferred procedure calls that are placed in a FIFO task queue, and allow the application to defer some computations until a later time. The TinyOS scheduler sequentially executes the tasks from the queue, and a task is run to completion and cannot be preempted by another task. However, hardware interrupts trigger the event-handlers and can preempt a task. When an interrupt occurs, the corresponding event-handler performs the minimum amount of computation and (may) post a task to the task queue for further computation.

The task queue in TinyOS is a circular buffer of function pointers. When a task is posted, TinyOS puts the task's function pointer in the next free slot of the buffer.

Figure 5.5-a shows the code snippet generated by avrgcc (for AVR platform) to put the function pointer of the task in the task queue. First, the address of the task is loaded into the register pair $r24, r25$. Then the function *TOS_post* is called, which puts the contents of the registers $r24, r25$ in the task queue. If the position of the task is shifted due to a change in the software, the first three instructions in this code snippet change between the two versions of the software. Note that the third statement is a function call statement and is handled by Zephyr as explained in Section 5.1. But the first two statements use function pointers.

The concept of indirection can be used here to handle function pointers as shown in Figure 5.5-b. The first two load statements are replaced with a jump to the fixed location *call loc_m* in the indirection table. The registers are loaded with the function address (contained in the indirection table) of the task. Finally the the flow of control is directed back to the line following *call loc_m* by the use of the *ret* statement. In this way, we preserve the similarity between the two versions of the software in all parts of the code except the indirection table. In the case where the task is posted to the task queue multiple times, this technique minimizes the change. If the task is posted n times during program execution, there would be n changes between the old and the new versions in the baseline case. With Zephyr, however, there will be a single change (in the indirection table) regardless of the number of times the task is posted to the task queue.

For a wide range of software change cases that we experimented with (as explained in Section 5.6), we find that a given task is generally posted only once in the program. Only very few tasks are posted more than once (typically less than 5 times). On the other hand, a given function is typically called multiple times in the program. As a result, there is not much reduction in the size of the delta script by using this technique for handling function pointers. Hence in our experiments, we do not perform the change depicted in Figure 5.5. However, if the user application has many tasks each of which are posted multiple times in the program, then this technique could be used to further reduce the size of the delta script.

5.4 Metacommands for common patterns of changes

After the delta script is created through the above mentioned techniques, Zephyr scans through the script file to identify some common patterns and applies the following optimizations to further reduce the delta size.

5.4.1 CWI command

In many cases, the delta script has the following sequence of commands:

```
COPY <oldOffset=O1> <len=L1> <newOffset=N1>
INSERT <newOffset> <len=l1> <data1>
COPY <oldOffset=O2> <len=L2> <newOffset=N2>
INSERT <newOffset> <len=l1> <data2>
COPY <oldOffset=O3> <len=L3> <newOffset=N3>
INSERT <newOffset> <len=l1> <data3>
```

and so on. Let L_i indicate a large value, and l_i a small value. Here, small INSERT commands are present in between large COPY commands. Here we have COPY commands that copy large chunks of size *L1, L2, L3,* ... from the old image followed by INSERT commands with very small values of *len= l1*. Further *O1+L1+l1=O2*, *O2+L2+l1=O3*, and so on. In many software change cases that we evaluated, we found that two blocks in two versions of the image match perfectly, except at few places where a single byte operand of some instructions differ. In other words, if the blocks corresponding to INSERT commands with small *len* had matched, we would have obtained a very large superblock. So Zephyr replaces such sequences with the COPY_WITH_INSERTS (CWI) command.

```
CWI <oldOffset=O1> <newOffset=N1>
<len=L1+l1+...+Ln> <dataSize=l1>
<numInserts=n> <addr1> <data1>
<addr2> <data2> ... <addrn> <datan>
```

Here *dataSize=l1* is the size of *datai (i=1,2..., n), numInserts=n* is the number of *(addr,data)* pairs, *datai* are the data that have to be inserted in the new image at the offset *addri*. This command tells the sensor node to copy the *len=L1+l1+...+Ln* number of bytes of data from the old image at offset *O1* to the new image at the offset *N1*, but to insert *datai* at the offset *addri (i=1, 2, ..., n)*.

5.4.2 REPEAT command

This command is useful for reducing the number of bytes in the delta script that are needed to transfer the indirection table. As shown in Figure 5.4-b, the indirection table consists of the pattern *call fun1, ret, call fun2, ret , ...* where the same string of bytes (say *S1 = ret; call*) repeats, with only the addresses for *fun1, fun2*, etc. changing between them. So Zephyr uses the following command to transfer the indirection table.

```
REPEAT <newOffset> <numRepeats=n>
<addr1> <addr2> ... <addrn>
```

This command puts the string *S1* at offset *newOffset* in the new image followed by *addr1*, then *S1*, then *addr2*, and so on till *addrn*. Note that the CWI command could have also been used for this case, but since string *S1* is fixed, fewer bytes are needed using the REPEAT command. This optimization is not applied if the addresses of the call instructions match in the indirection tables of the old and new images. In that case, the COPY command is used to transfer identical portions of the indirection table.

5.4.3 No offset specification

We note that if we build the new image on the sensor nodes in a *monotonic* order, then Zephyr does not need to specify the offset in the new file in any of the above commands. Monotonic means Zephyr always writes at location x of the new image

before writing at location y, for all $x < y$. Instead of the new offset being provided, a counter is maintained and incremented as the new image is built, and the next write always happens at this counter, allowing the *newOffset* field to be dropped from all the commands.

We find that for Case 2, where some functions are shifted due to the addition of few lines in the software, the delta script produced with the application-level modifications is 280 bytes compared to 2183 bytes when optimized Rsync is used without application-level modifications. The size of the delta script without the metacommands is 528 bytes. This illustrates the importance of application-level modifications in reducing the size of the delta script and making it consistent with the amount of actual change made in the software.

5.5 Delta distribution stage

One of the factors that we considered for the delta distribution stage was to have as small a delta script as possible even in the worst case when there is a huge change in the software. In this case there is little similarity between the old and the new code images, and the delta script basically consists of a large INSERT command to insert almost the entire binary image. To have a small delta script even in such extreme cases, it is necessary that the binary image itself be small. Since the size of the binary image transmitted by Stream [88] is almost half the size of that of Deluge [92], Zephyr uses the approach from Stream, with some modifications for wirelessly distributing the delta script. The core data dissemination method of Stream is the same as in Deluge. Deluge uses a monotonically increasing version number, segments the binary code image into pages, and pipelines the different pages across the network. The code distribution occurs through a three-way handshake of advertisement, request, and code broadcast between neighboring nodes. Unlike Deluge, Stream does not transfer the entire reprogramming component every time a code update is done. The reason for this requirement in Deluge is that the reprogramming component needs to

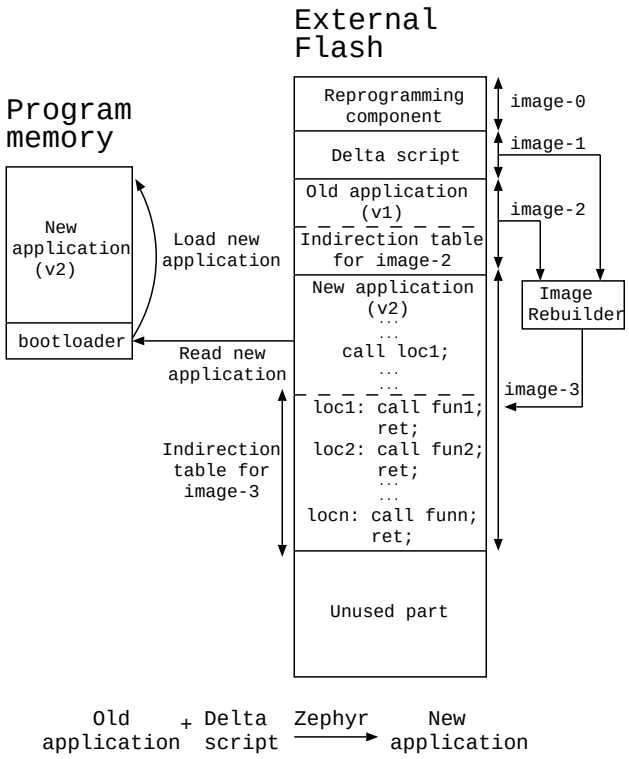


Fig. 5.6. Image rebuild and load stage. The right side shows the structure of external flash in Zephyr.

be running on the sensor nodes all the time so that the nodes can be receptive to future code updates and these nodes are not capable of multitasking (running more than one application at a time). Stream solves this problem by storing the reprogramming component in the external flash and running it on demand—whenever reprogramming is to be done.

Distinct from Stream, Zephyr divides the external flash as shown in the right side of Figure 5.6. The reprogramming component and delta script are stored as image 0 and image 1 respectively. Image 2 and image 3 are the user applications—one old version and the other current version which is created from the old image and the delta script as discussed in Section 5.5.1. The protocol works as follows:

1. Let image 2 be the current version (v_1) of the user application. Initially all nodes in the network are running image 2. At the host computer, delta script is generated between the old image (v_1) and the new image (v_2).

2. The user gives the command to the base node to reboot all nodes in the network from image 0 (i.e. the reprogramming component).

3. The base node broadcasts the reboot command and itself reboots from the reprogramming component.

4. The nodes receiving the reboot command from the base node rebroadcast the reboot command and themselves reboot from the reprogramming component. This is controlled flooding because each node broadcasts the reboot command only once. Finally, all nodes in the network are executing the reprogramming component.

5. The user then injects the delta script into the base node. It is wirelessly transmitted to all nodes in the network using the usual 3-way handshake of advertisement, request, and code broadcast, as in Deluge. Note that unlike Stream and Deluge, which transfer the application image itself, Zephyr transfers only the delta script.

6. All nodes store the received delta script as image 1.

As mentioned above, Zephyr uses a simple flooding scheme to send reboot commands to all nodes in the network. This method is generally sufficient to ensure that all nodes receive the reboot command. Note that every node broadcasts the reboot command once, before rebooting from the reprogramming component (i.e. img-0 as shown in Figure 5.6). So, even if a node does not receive the reboot command from one of its neighbors due to, say, poor link quality between them, it may receive the command from other neighbor(s). However, due to very poor link reliability with all neighbors, or hardware/software faults, some nodes may not be able to receive

the reboot command. To address this issue, Zephyr provides "eventual consistency" property, similar to Stream [88].

Let us call the nodes which do not receive reboot command from any of its neighbors the *faulty nodes*. In Zephyr, sensor nodes use Trickle [107] algorithm to periodically advertise their metadata consisting of the version number of the program images that they possess (img-0 through img-3 in Figure 5.6). When the faulty nodes eventually recover from their faults, they will receive the advertisement message from their neighbors and learn that they do not possess the latest version of the binary image. This causes the faulty node to broadcast its advertisement message immediately and reboot from the reprogramming component (img-0). If the neighbors of the faulty node are executing img-0 (this happens if they have not yet completed downloading the new version of the application image), the faulty node starts downloading the new image from its neighbors according to the Deluge algorithm. If the neighbors of the faulty node are not executing img-0 (this happens if they have already downloaded the new version of the application image and have already started running the new image), they find that their faulty neighbor is not up-to-date and hence reboot from img-0 to bring the faulty node up-to-date. Then the faulty node downloads the delta and and builds the new image. Note that due to inherent failure-prone nature of sensor networks, many prior works [88, 92, 107] have shown that eventual consistency is a satisfactory solution. Hence Zephyr follows this approach.

5.5.1 Image rebuild and load stage

After the nodes download the delta script, they rebuild the new image using the script (stored as image 1 in the external flash) and the old image (stored as image 2 in the external flash). The image rebuilder stage consists of a *delta interpreter* which interprets the COPY, INSERT, CWI, and REPEAT commands of the delta script and creates the new image which is stored as image 3 in the external flash.

The methods of rebooting from the new image are slightly different in Stream and Zephyr. In Stream, a node automatically reboots from the new code once the code update has completed and it has satisfied all other nodes that depend on this node to download the new code. This means that different nodes in the network begin executing the new version of the code at different times. However, for Zephyr, we modified Stream so that all the nodes reboot from the new code after the user manually sends the reboot command from the base station (as in Deluge). We made this change because in many software change cases, the size of the delta script is so small that a node (say n_1) nearer to the base station quickly completes downloading the code before a node (say n_2) further away from the base station even starts requesting packets from n_1. As a result, n_1 reboots from the new code so fast that n_2 cannot even start the download process. Note that this does not, however, pose a correctness issue. After n_1 reboots from the new code, it will switch again to the reprogramming state when it receives an advertisement from n_2. This, however, incurs the performance penalty of rebooting from a new image. Our design choice has a good consequence —all nodes come up with the new version of the software at the same time. This avoids the situation where different nodes in the network run different versions of the software. When a node receives the reboot command, its bootloader loads the new software from image 3 of the external flash to the program memory (Figure 5.6). In the next round of reprogramming, image 3 will become the old image and the newly built image will be stored as image 2. As we will show later, the time to build the image is negligible compared to the total reprogramming time.

5.5.2 Dynamic page size

Stream divides the binary image into fixed-sized pages. The remaining space in the last page is padded with all 0s. Each page consists of 1104 bytes (48 packets per page with 23 bytes payload in each packet). With Zephyr, it is likely that in many cases, the size of the delta script will be much smaller than 1104 bytes. For

example, we have delta script of sizes of 17 bytes and 280 bytes for Case 1 and Case 2 respectively. Also, as we will show in Section 5.6.2, during the natural evolution of the software, it is more likely that the nature of the changes will be small or moderate and as a result, delta scripts will be much smaller than the standard page size. After all, the basic assumption behind any incremental reprogramming protocol is that in practice, the software changes are generally small and the similarities between the two versions of the software can be exploited to send only small delta. When the size of the delta script is much smaller than the page size, it is wasteful to transfer the whole page. So, we change the basic Stream protocol to use dynamic page sizes.

When the delta script is being injected in to the base node, the host computer informs it of the delta script size. If it is less than the standard page size, the base node includes this information in the advertisement packets that it broadcasts. When other nodes receive the advertisement, they also include this information in the advertisement packets that they send. As a result, all nodes in the network know the size of the delta script and they make the page size equal to the actual delta script size. So unlike Deluge or Stream which transmit all 48 data packets per page, Zephyr transmits only the required number of data packets if the delta script size is less than 1104 bytes. Note that the granularity of this scheme is the packet size, i.e., the last packet of the last page may be padded with zeros. But this results in small enough wastage that we did not feel justified in introducing the additional complexity of dynamic packet sizes. Our scheme can be further modified to advertise the actual number of packets in the last page to minimize the wastage. For example in the case where the delta script has 1105 bytes, it would transfer two pages, the first page with 48 packets and the second with 1 packet.

5.6 Experiments and results

In order to evaluate the performance of Zephyr, we consider a number of software change scenarios. The software change cases for standard TinyOS applications that we consider are as follows:

Case 1: Change the Blink application from blinking a green LED every second to blinking it every 2 seconds.

Case 2: Add a few lines added to the Blink application.

Case 3: Change the Blink application to CntToLedsAndRfm: CntToLedsAndRfm is an application that displays the lowest 3 bits of the counting sequence on the LEDs as well as sends them over radio.

Case 4: Change CntToLeds to CntToLedsAndRfm: CntToLeds is the same as CntToLedsAndRfm except that the counting sequence is not transmitted over radio.

Case 5: Change Blink to CntToLeds.

Case 6: Change Blink to Surge: Surge is a multi hop routing protocol. This case corresponds to a complete change in the application.

Case 7: Change CntToRfm to CntToLedsAndRfm: CntToRfm is the same as CntToLedsAndRfm except that the counting sequence is not displayed on the LEDs.

In order to evaluate the performance of Zephyr with respect to natural evolution of the real world software, we considered a real world sensor network application called eStadium [130] which is deployed in Ross Ade football stadium at Purdue University. eStadium applications provide safety and security functionality, infotainment features such as coordinated cheering contests among different parts of the stadium using the microphone data, information to fans about lines in front of concession stands, and so forth. We considered a subset of the changes that the software had actually gone through, during various stages of refinement of the application.

Case A: Change an application that samples battery voltage and temperature from an MTS310 [1] sensor board to one where a few functions are added to also sample the photo sensor.

Case B: During the deployment phase, we decided to use opaque boxes for the sensor nodes. So, a few functions were deleted to remove the light sampling features.

Case C: In addition to temperature and battery voltage, we added the features for sampling all the sensors on the MTS310 board except light (e.g., microphone, accelerometer, magnetometer). This was a huge change in the software with the addition of many functions. For accelerometer and microphone, we collected mean and mean square values of the samples taken during a user-specified window size.

Case D: This is the same as Case C but with addition of few lines of code to get microphone peak value over the user-specified window size.

Case E: We decided to remove the feature of sensing and wirelessly transmitting to the base node, the microphone mean value since we were interested in the energy of the sound which is given by the mean square value. A few lines of code were deleted for this change.

Case F: This is same as Case E except we added the feature of allowing the user to put the nodes to sleep for a user-specified duration. This was also a huge change in the software.

Case G: We changed the microphone gain parameter. This is a simple parameter change.

We can group the above changes into 4 classes:

Class 1 (Small change SC): This includes Case 1 and Case G where only a parameter of the application was changed.

Class 2 (Moderate change MC): This includes Case 2, Case D, and Case E. They consist of the addition or deletion of few lines of the code.

Class 3 (Large change LC): This includes Case 5, Case 7, Case A, and Case B where few functions are added or deleted or changed.

Class 4 (Very large change VLC) : This includes Case 3, Case 4, Case 6, Case C, and Case F, where the software has changed completely (the goal of the software has changed). For example, in Case 6, the goal of the software is changed from periodically blinking an LED to perform routing.

These classes of software changes are based on the degree of the change that the software has undergone at the application level. Many of the above cases involve changes in the OS kernel as well. Strictly speaking, in TinyOS, there is no separation between the OS kernel and the application. The two are compiled as one large monolithic image that is run on the sensor nodes. So, if the application is modified such that new OS components are added or existing components are removed, then the delta generated would include OS updates as well. For example, in Case C we change the application that samples temperature and battery voltage to the one that samples microphone, magnetometer and accelerometer sensors in addition to temperature and battery. This causes new OS components to be added—the device drivers for the added sensors.

5.6.1 Block size for byte-level comparison

We modified Jarsync [133], a java implementation of the Rsync algorithm, to achieve the optimizations mentioned in Section 4.2. From here onward, by "semi-optimized Rsync", we mean the scheme that combines two or more contiguous matching blocks into one super-block. It does not necessarily produce the maximal super-block. By "optimized Rsync" we mean our scheme that produces the maximal super-block but without the application-level modifications.

As shown in Figure 5.7, the size of the delta script produced by either Rsync or optimized Rsync depends on the block size used in the algorithm. Recollect that the comparison is done at the granularity of a block. As expected, Figure 5.7 shows that the size of the delta script is largest for Rsync and smallest for optimized Rsync. Figure 5.7 also shows that as the block size increases, the size of the delta script produced by Rsync and semi-optimized Rsync decreases till a certain point after which it has an increasing trend. The size of the delta script depends on two factors: 1) the number of commands in the delta script and 2) the size of the data in the INSERT command. For Rsync and semi-optimized Rsync, for block size below the minima

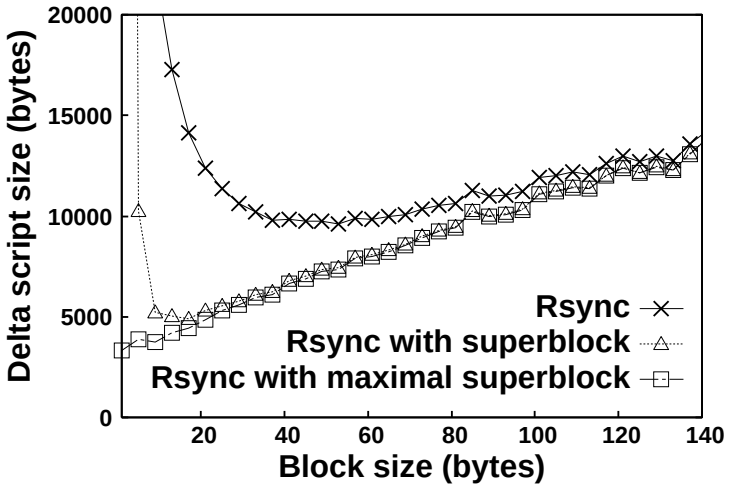


Fig. 5.7. Delta script size versus block size

point, the number of commands is high because these schemes find lots of matching blocks but not (necessarily) the maximal super-block. As block size increases in this region, the number of matching blocks and hence the number of commands drops sharply, causing the delta script size to decrease. However, as the block size increases beyond the minima point, the decrease in the number of commands in the delta script is dominated by the increase in the size of new data to be inserted. As a result, the delta script size increases.

For optimized Rsync, there is a monotonic increasing trend for the delta script size as block size increases. There are, however, some small oscillations in the curve, as a result of which the optimal block size is not always one byte. The small oscillations are because increasing the block size decreases the size of maximal super-blocks and increases the size of data in INSERT commands. But sometimes the small increase in the size of the data can contribute to reducing the size of the delta script by reducing the number of COPY commands. Nonetheless, there is an overall increasing trend for optimized Rsync. This has the important consequence that a system administrator using Zephyr does not have to figure out the block size to use in uploading code for each application change. She can use the smallest or close to smallest block size and let Zephyr be responsible for compacting the size of the delta script. In all

further experiments, we use the block size that gives the smallest delta script for each scheme—Rsync, semi-optimized Rsync, and optimized Rsync.

5.6.2 Size of delta script

The goal of an incremental reprogramming system is to reduce the size of the delta script that needs to be transmitted to the sensor nodes. A small delta script translates to less reprogramming time and energy due to fewer packet transmissions over the network and small number of external flash writes on the node. Figure 5.8 and Table 5.1 compare the delta script produced by Deluge, Stream, Rsync, semi-optimized Rsync, Optimized Rsync, and Zephyr. Table 5.1 also compares Zephyr with *ZephyrWOMetaCmds*—the case where all application level modifications are used except meta commands. For Deluge and Stream, the size of the information to be transmitted is the size of the binary image, while for the other schemes it is the size of the delta script. Deluge, Stream, Rsync, and semi-optimized Rsync take up to 1987, 1324, 49, and 6 times more bytes than Zephyr, respectively. Note that for cases belonging to moderate or large change, the application level modifications of Zephyr contribute to significantly reducing the size of delta script compared to optimized Rsync. Optimized Rsync takes up to nine times more bytes than Zephyr. These cases correspond to function shifts in the software. As a result, application-level modifications have great effect in those cases. In practice, these are probably the most frequently occurring categories of changes in the software. Case 1 and Case G are parameter change cases which do not shift any function. As a result, we find that delta scripts produced by optimized Rsync without application-level modifications are only slightly larger than the ones produced by Zephyr. Also, even for very large software change cases (like cases 6, F, and C), Zephyr is more efficient compared to other schemes. In summary, application-level modifications have the greatest effects in moderate and large software change cases, a significant effect in the very large

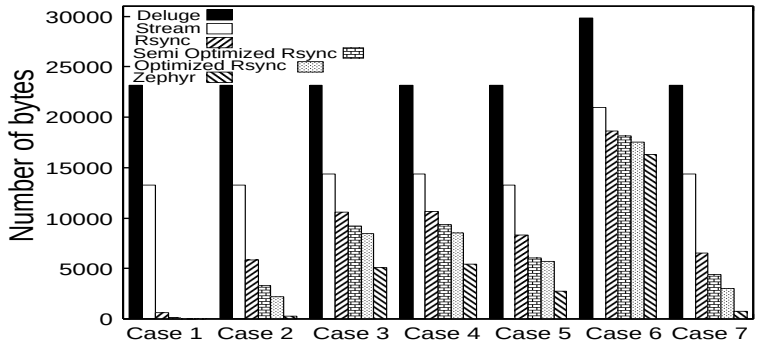
Table 5.1
Comparison of delta script size of various approaches. Deluge, Stream and Rsync represent prior work.

	Deluge:Zephyr	Stream:Zephyr	Rsync:Zephyr	SemiOptRsync:Zephyr	OptRsync:Zephyr	ZephyrWOMetaCmds:Zephyr
Case 1	1400.82	779.29	35.88	6.47	1.35	1.35
Case 2	85.05	47.31	20.81	11.75	7.79	1.99
Case 3	4.52	2.80	2.06	1.80	1.64	1.38
Case 4	4.29	2.65	1.96	1.72	1.57	1.30
Case 5	8.47	4.84	3.03	2.22	2.08	1.39
Case 6	1.83	1.28	1.14	1.11	1.07	1.05
Case 7	29.76	18.42	8.34	5.61	3.87	1.52
Case A	7.60	5.06	3.35	2.66	2.37	1.6
Case B	7.76	5.17	3.38	2.71	2.37	1.61
Case C	2.63	1.82	1.50	1.39	1.35	1.16
Case D	203.57	140.93	36.03	14.36	7.84	2.33
Case E	243.25	168.40	42.03	17.66	9.01	2.43
Case F	2.75	1.83	1.50	1.36	1.33	1.18
Case G	1987.2	1324.8	49.6	6.06	1.4	1.4

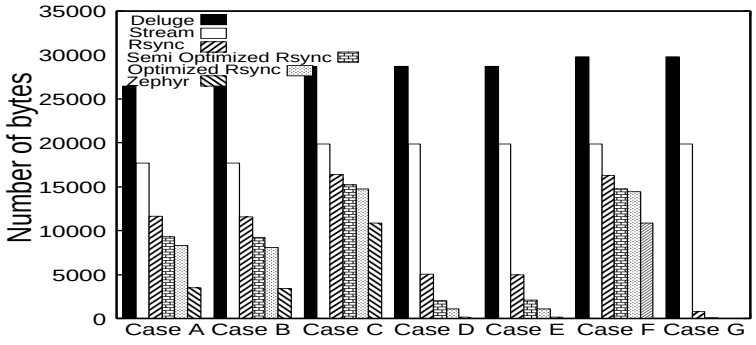
software change case (in terms of absolute delta size reduction) and a small effect on the very small software change cases.

Comparison with other incremental approaches: Rsync represents the algorithm used by Jeong and Culler [121] to generate the delta by comparing the two executables without any application-level modifications. We find that [121] produces up to 49 times larger delta script than Zephyr. As mentioned above, in [122]'s approach, the program memory is fragmented and used less efficiently than in Zephyr. Flexcup [123], though capable of incremental linking and loading on TinyOS, generates high traffic through the network because of large symbol and relocation tables. Also, Flexcup is implemented only on an emulator whereas Zephyr runs on the real sensor node hardware.

To describe function shifts, [87] augments the delta script with a *patch list command*—a list of {*begin address, end address, offset*}-tuples, each of which describes the offset by which each function is shifted in the new version of the user application.



(a) TinyOS software change cases



(b) eStadium software change cases

Fig. 5.8. Size of data transmitted for reprogramming

More specifically, this command says that all functions which lie between {*begin address, end address*} are shifted by the *offset* bytes in the new version. The patch list command incurs an overhead of $2 + 6 * count$ bytes (2 bytes for *count*—the number of patch lists; 2 bytes each for *begin address, end address, and offset*). The patch list command is useful for small software change scenarios where many functions are shifted by the same offset and thus can be described by a few patch lists. However, for many practical software change cases, different functions are shifted by different offsets (e.g. software change cases 3, 4, 5, 6, 7, B, C, and F in our experiments). In such cases, the overhead due to the patch list command can be significant because

of the increase in the number of patch lists required to express the shifts in all the functions between the old and the new versions of the code.

The patch list command can also significantly increase the size of the delta script (although patch list command can be thought of as a part of the delta script, we categorize COPY, INSERT commands as delta script, and patch lists as patch list command, for convenience of explanation). Note that in [87], when executing a COPY command, a node checks for each word that it copies if the previous word is the opcode of a patchable instruction—the instruction that uses function address. If so, and if the copied word lies in the range covered by one of the entries in the patch list, the node adds the corresponding offset to the copied word. However, implementing this is difficult as well as architecture-dependent. This is because it is not always possible to distinguish a patchable instruction by just looking at the opcode in the binary executable. For example, while pushing a task in the task queue, TinyOS uses *ldi* statements to load registers *r24* and *r25* with the address of the task (a function). By just looking at the binary code, it is not possible to say whether the operand of *ldi* instruction is a function address or say some constant. As a result, implementation in [87] does not look for patchable instruction. Instead it patches all binary words that lie in the {*begin address, end address*} range. As a consequence, the node ends up performing many "mispataches" as acknowledged by the authors. To correct these mispatches, [87] uses REPAIR commands (these are used for other purposes also) in the delta script, thereby increasing the size of the delta script. This is illustrated by the fact that for most of the software change cases used in [87], the decrease in the size of the delta script is not as significant as in our experiments. Furthermore, [87] assumes that the function address is always preceded by an instruction opcode (for simplifying the identification of the patchable instruction). This, however, is dependent on the microcontroller architecture and may not be true for many architectures. For example, in ATmega128, as mentioned above, the second operand of the *ldi* statement can be the function address. We believe that Zephyr offers an elegant solution without all these complications and overhead.

Apart from these core differences, [87] does not present an implementation of a complete incremental reprogramming system. They focus mainly on the generation of the difference, and not on actual dissemination. Zephyr, on the other hand, is a complete usable incremental reprogramming system. Furthermore, [87] presents an evaluation of their system with very few software change scenarios, whereas in this chapter, we present a comprehensive evaluation of Zephyr.

In the software change cases that we considered, the time to compile, link (with the application-level modifications) and generate the executable file was at most 2.85 seconds, and the time to generate the delta script using optimized Rsync was at most 4.12 seconds on a 1.86 GHz Pentium processor. These times are negligible compared to the time to reprogram the network, for any but the smallest of networks. Furthermore, these times can be made smaller by using more powerful server-class machines. TinyOS applies extensive optimizations on the application binaries to run it efficiently on the resource-constrained sensor nodes. One of these optimizations involves inlining of several (small) functions. We do not change any of these optimizations. In systems which do not inline functions, Zephyr's advantage will be even greater since there will be more function calls. Zephyr's advantage will be minimal if the software change does not shift any function. For such a change, the advantage will be only from the optimized Rsync algorithm. But such software changes are very rare, e.g. when only the values of the parameters in the program are changed. Any addition/deletion/modification of the source code in any function except the one which is placed at the end of the binary will cause all following functions to be shifted.

5.6.3 Testbed experiments

We perform testbed experiments using Mica2 [1] nodes for grid and linear topologies. For the grid network, the transmission range R_{tx} of a node is set such that $\sqrt{2}d < R_{tx} < 2d$, where d is the separation between the two adjacent nodes in any row or column of the grid. The linear networks have the nodes with R_{tx} such that

Table 5.2
Ratio of reprogramming times of other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	22.39	48.9	32.25	25.04	48.7	30.79	14.89	33.24	17.42	1.92	3.08	2.1
Stream:Zephyr	14.06	27.84	22.13	16.77	40.1	22.92	10.26	20.86	10.88	1.54	2.23	1.46
Rsync:Zephyr	1.03	8.17	2.55	5.66	12.78	8.07	5.22	10.89	6.50	1.34	1.71	1.42
Optimized Rsync:Zephyr	1.01	1.1	1.03	2.01	4.09	2.71	2.05	3.55	2.54	1.27	1.55	1.35

$d < R_{tx} < 2d$, where d is the distance between the adjacent nodes. Due to fluctuations in transmission range, occasionally a non-adjacent node will receive a packet. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped, to achieve a truly multi-hop network. This kind of software topology control has been used in other works also [50,134]. A node situated at one corner of the grid or end of the line acts as the base node. We provide a quantitative comparison of Zephyr with Deluge [92], Stream [88], Rsync [121] and optimized Rsync without application-level modifications. Note that Jeong and Culler [121] reprogram only nodes within one hop of the base node, but we used their approach on top of a multi-hop reprogramming protocol to provide a fair comparison. The metrics for comparison are reprogramming time and energy. We perform these experiments for grids of size 2x2 to 4x4 and linear networks of size 2 to 10 nodes. We choose four software change cases, one from each equivalence class: Case 1 for Class 1 (SC), Case D for Class 2 (MC), Case 7 for Class 3 (LC), and Case C for Class 4 (VLC). Note that in the evaluations that follow, Rsync refers to the approach by Jeong and Culler [121].

Reprogramming time

Time to reprogram the network is the sum of the time to download the delta script and the time to build the new image. The time to download the delta script is the time interval between the instant t_0, when the base node sends the first advertisement

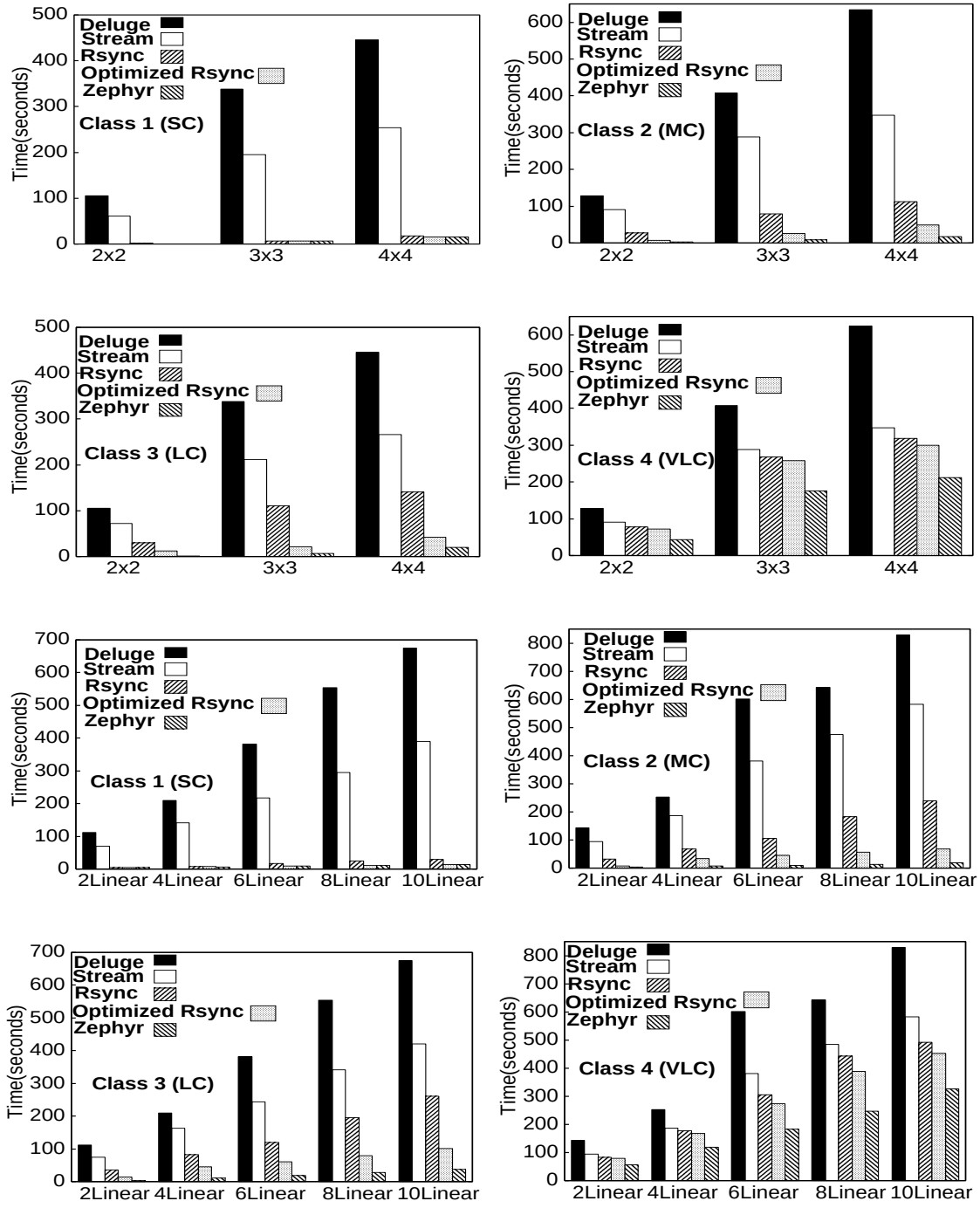


Fig. 5.9. Comparison of reprogramming times for grid and linear networks.

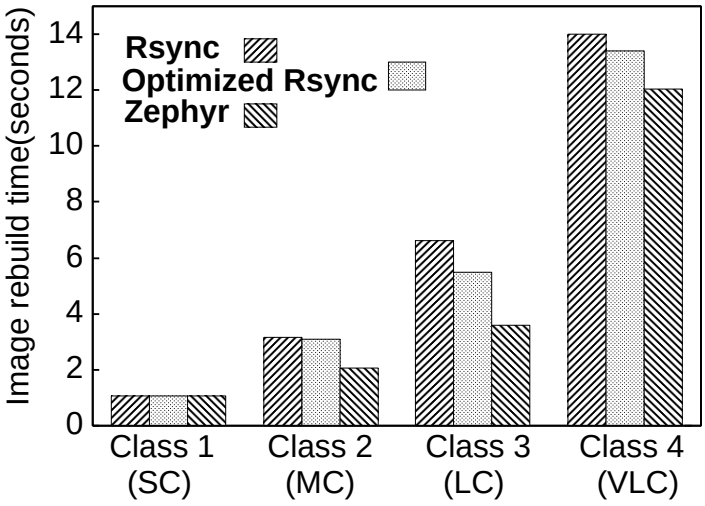


Fig. 5.10. Time to rebuild image on the sensor node.

packet, to the instant t_1 when the last node (the one that takes the longest time to download the delta script) completes downloading the delta script. Since clocks maintained by the nodes in the network are not synchronized, we cannot take the difference between the time instant t_1 measured by the last node and t_0 measured by the base node. To solve this synchronization problem, we use the approach of [50], which achieves this with minimal overhead traffic.

Figure 5.9 compares reprogramming times of other approaches with Zephyr for different grid and linear networks. Table 5.2 compares the ratio of reprogramming times of other approaches to Zephyr. It shows minimum, maximum and average ratios over these grid and linear networks. As expected, Zephyr outperforms non-incremental reprogramming protocols like Deluge and Stream significantly for all the cases. Zephyr is also up to 12.78 times faster than Rsync, the approach of Jeong and Culler [121]. This illustrates that the Rsync optimization and the application-level modifications of Zephyr are important in reducing the time to reprogram the network. Zephyr is also significantly faster than optimized Rsync without application-level modifications for moderate, large, and very large software changes. In these cases, the software changes cause function shifts. So, these results show that application level modifica-

tions greatly mitigate the effect of function shifts and reduce the reprogramming time significantly. For the small change case where there are no function shifts, Zephyr, as expected, is only marginally faster than optimized Rsync without application-level modifications. In this case, the size of the delta script is very small (17 and 23 bytes for Zephyr and optimized Rsync respectively) and hence there is little room for improvement. Since Zephyr transfers less information at each hop, Zephyr's advantage will increase with the size of the network. Figure 5.10 shows the time to rebuild the new image on a node. It increases with the increase in the scale of the software change, but is negligible compared to the total reprogramming time.

Reprogramming energy

The most important factors that contribute to energy consumption during reprogramming are radio transmissions, receptions, idle listening, and flash writes. By idle listening, we mean the state when a node turns on its radio transceiver, but does not transmit or receive any packet. Obviously, the energy cost due to radio transmissions and receptions are directly proportional to the number of packets that are transmitted by all nodes in the network for reprogramming. As mentioned earlier, the downloaded delta script is first stored in the external flash by each sensor node. The new image, which is built using the delta script and the old image, is also stored in the external flash. Then the new image is loaded from from external flash to the flash program memory by the bootloader. Thus the number of flash program memory accesses for loading the new image from external flash is independent of the number of packets received by the sensor node, and is same for all reprogramming protocols because ultimately each protocol is creating the same new version of the program. However, the number of external flash accesses (to store the delta script) is directly proportional to the number of packets received by the sensor node. Thus, the total number of packets transmitted by all nodes in the network during reprogramming is a good measure of the energy cost due to radio transmissions and receptions, as well

as flash writes. In this section, we first compare different protocols in terms of total number of packets transmitted by all nodes in the network for reprogramming. Later we will analyze idle listening energy cost.

Table 5.3
Ratio of number of packets transmitted during reprogramming by other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	90.01	215.39	162.56	40	204.3	101.12	12.27	55.46	25.65	2.51	2.9	2.35
Stream:Zephyr	53.76	117.92	74.63	28.16	146.1	82.57	8.6	36.19	15.97	1.62	2.17	1.7
Rsync:Zephyr	2.47	7.45	5.38	6.66	38.28	21.09	3.28	12.68	6.69	1.50	1.78	1.60
Optimized Rsync:Zephyr	1.13	1.69	1.3	4.38	22.97	9.47	2.72	10.58	3.95	1.38	1.64	1.49

Figure 5.11 and Table 3 compare the number of packets transmitted by Zephyr with other schemes for grid and linear networks of different sizes. The number of bytes transmitted by all nodes in the network for reprogramming by Deluge, Stream, Rsync, and optimized Rsync is up to 215, 146, 38, and 22 times more than that by Zephyr. The fact that *Rsync:Zephyr>1* indicates that Zephyr is more energy efficient than the incremental reprogramming approach of [121]. The application-level modifications are significant in reducing the number of packets transmitted by Zephyr compared to optimized Rsync without such modifications. Note that in cases like Case 7 and Case D (moderate to large change class), application-level modifications have the greatest impact where the functions get shifted. Application-level modifications preserve maximum similarity between the two images in such cases, thereby reducing the reprogramming traffic overhead. In cases where only some parameters of the software change without shifting any function, the application-level modifications achieve a smaller reduction. But the size of the delta is already very small and hence reprogramming is not resource intensive in these cases. Even for very large software changes, Zephyr significantly reduces the reprogramming traffic.

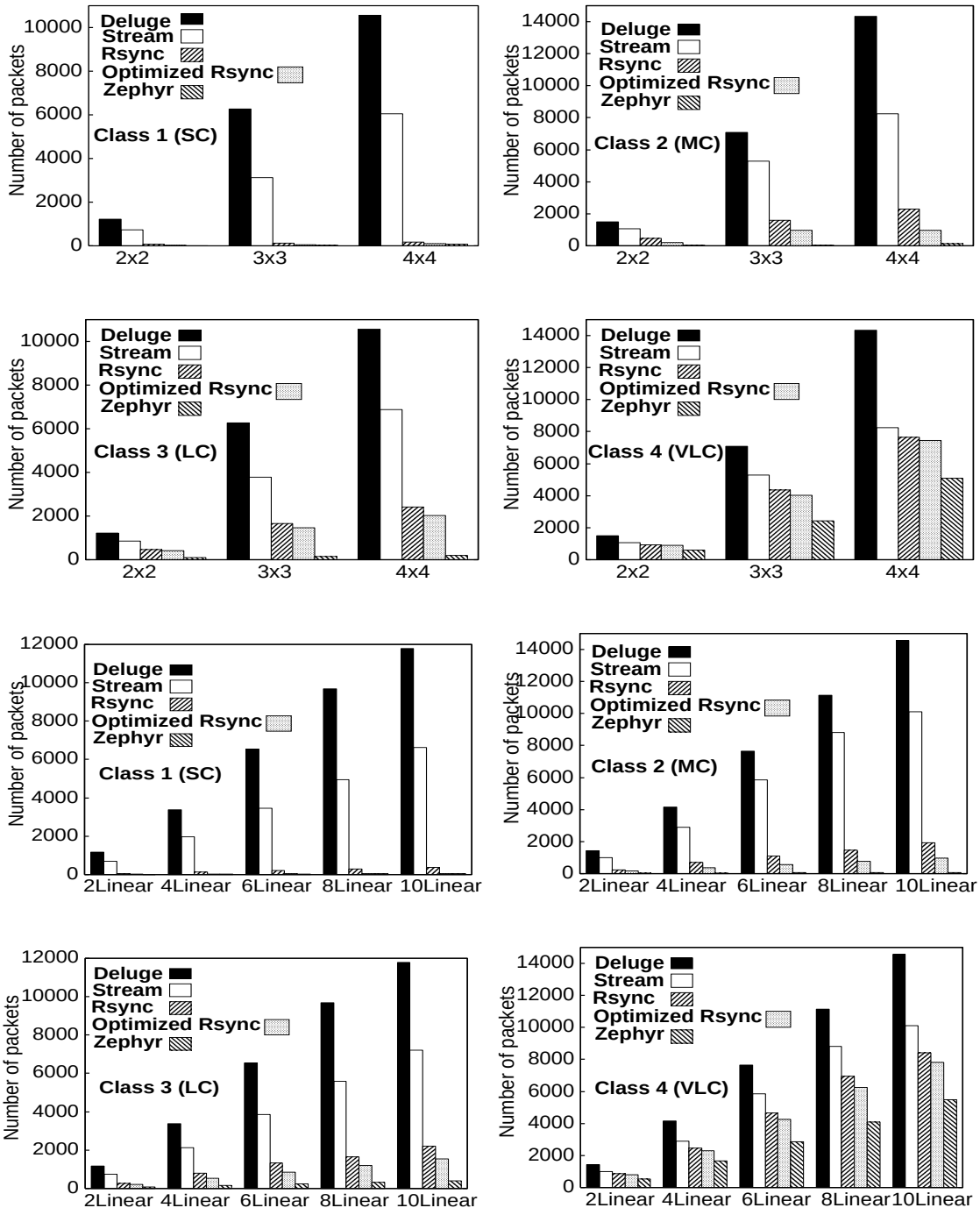


Fig. 5.11. Comparison of number of packets transmitted during reprogramming.

As mentioned above, radio communication is a major source of energy consumption during reprogramming. The energy cost due to radio communication can be grouped into 3 categories: 1) Idle listening energy cost, E_1: It is the energy consumption due to nodes listening to the wireless medium when there is no packet transmission in their neighborhood. 2) Energy cost due to transmission and reception of *unnecessary* packets, E_2: By unnecessary packets, we mean a) the corrupt packets, b) code packets that a node has already downloaded, and c) the code packets which a node does not store upon reception. Note that in Zephyr, like in Deluge, a node downloads code packets in a *monotonic order*—it downloads packets of a page x before packets of page y, for all $x < y$. This is done to avoid the state maintenance overhead. 3) Energy cost due to transmission and reception of *necessary* packets, E_3: By necessary packets, we mean the code packets that a node stores upon reception.

The underlying MAC protocol affects energy costs E_1, E_2, and E_3. Low power listening (LPL) MAC protocols [135–137] cause the sensor nodes to put the radio transceiver to sleep mode for most of the time and wake it up periodically for a short period of time to sample the channel to check if there is any radio transmission in their neighborhood. Idle listening energy cost is significantly reduced by duty-cycling LPL MACs because the time duration for which the radio is turned on without any packet transmission or reception is limited to short channel sampling periods. The asynchronous LPL MACs (like BMAC [136], XMAC [135]) avoid the need for time synchronization among nodes by having the transmitter transmit a (long) preamble sequence before each packet transmission such that the preamble duration is at least as long as the sleep period of the nodes[1]. The long preamble ensures that when a node wakes up to sample the channel, it is guaranteed to to hear the preamble. When a node hears a preamble, it turns its radio on until the packet is received. Thus the

[1]In some MAC protocols like XMAC, the preamble can be made shorter than than the sleep period for unicast packets. But for broadcast packets, the preamble should be at least as long as the sleep duration. Since most of the packet transmissions during reprogramming are of broadcast nature (except request packets of advertisement-request-data handshake), we assume that the preamble length is at least as long as the sleep period.

energy cost is incurred not only during packet transmission and reception, but also during preamble transmission and reception.

The energy costs E_2 and E_3 are due to transmission and reception of both preambles and actual packets (unnecessary packets for E_2 and necessary packets for E_3). Note that E_2 also includes the energy cost due to a node keeping its radio on after receiving a preamble for a packet that it later finds it is not interested in. Let N_P be the total number of packets transmitted by all nodes in the network during reprogramming. It is the sum of all necessary and unnecessary packets. Clearly, the energy costs E_2 and E_3 are directly proportional to N_P. Thus the total number of packets transmitted by all nodes in the network during reprogramming provides a measure of E_2 and E_3. From figure 5.11, we see that Zephyr incurs significantly low energy costs E_2 and E_3 compared to other protocols.

Each node spends some time (say t) in transmitting and receiving necessary and unnecessary packets. t also includes preamble duration. During rest of the time period (say, $T_R - t$, where T_R is the total reprogramming period), the node samples the channel periodically (without detecting radio transmission in its neighborhood) according to its duty-cycling schedule. Thus, the reprogramming period T_R provides a measure of idle energy cost E_1. Figure 5.9 shows that Zephyr incurs significantly less idle energy cost E_1 compared to other protocols.

Next we present a simplified mathematical analysis to demonstrate the qualitative argument that we presented above—the total number of packets transmitted by all nodes in the network is a measure of the energy costs E_2 and E_3, and the reprogramming period is a measure of the energy cost E_1. To simplify the analysis, we consider a single hop network. Let this network takes T_R time to be reprogrammed. Suppose that N_P is the total number of packets transmitted by all nodes in the network during reprogramming. The upper bound [2] of the time period t spent by each node in the

[2]This is an upper bound because a) a node may not turn its radio on for the entire preamble duration if it detects the preamble at an instant other than the exact start of the preamble, and b) during the periodic channel sampling, a node may miss the preamble because of bad link conditions.

network for receiving and transmitting necessary and unnecessary packets (including preamble) is

$$t = N_P * t_p \tag{5.1}$$

where t_p is the time to transmit a single packet. t_p is given by

$$t_p = t_{PR} + t_{DATA} \tag{5.2}$$

where t_{PR} is the time to transmit the preamble and t_{DATA} is the time to transmit the actual (data) packet. To calculate t_{PR} and t_{DATA}, let us consider XMAC, which is the LPL MAC used by TinyOS. As mentioned above, in LPL MAC schemes, before transmitting a packet, each node transmits a preamble which is at least as long as the sleep period to ensure that its neighbor receives the preamble and hence keeps its radio on to receive the actual data packet. In XMAC, the length of the preamble is reduced (for unicast packets) because during the preamble period, a sender node transmits a series of strobe packets containing the receiver's address with a gap between the successive strobe packets. When the receiver node wakes up and detects the strobe packet, it sends an ACK during the gap between two successive strobe packets. Upon receiving the ACK, the sender node stops transmitting the strobe packets and immediately sends the actual data packet. Note that during reprogramming, most of the packets are broadcast (except request packets). Hence, in XMAC, the preamble strobe packets have to be sent for the entire sleep period to make sure that all neighbors receive the broadcast packet. As a result, the preamble duration (t_{PR}) in XMAC is equal to the sleep period. The sleep period depends upon duty cycle and time required by the radio transceiver to sample the channel. Let t_{cs} be the channel sampling period. Since duty cycle is given by $d_c = t_{cs}/(t_{cs}+t_{PR})$, thus the preamble period is given by

$$t_{PR} = \frac{t_{cs}(1 - d_c)}{d_c} \tag{5.3}$$

t_{cs} should be long enough to ensure that it does not lie in the gap between the preamble strobe packets, and thus the receiver does not miss the preamble strobe packet. For this, t_{cs} should be slightly longer than the gap between the strobe packets. Let us approximate t_{cs} as follows:

$$t_{cs} = T_{ACK} + T_{TA} \tag{5.4}$$

where T_{ACK} is the time required by the receiver to transmit the ACK packet and T_{TA} is the turnaround time—time required by the radio transceiver to switch from receive mode to transmit mode. Time to transmit actual data packet depends on the size of the packet and the data rate supported by the radio transceiver. Assuming packet size=36 bytes (the default maximum packet size in TinyOS) and data rate=250kbps (for IEEE 802.15.4 based radios like CC2420), time to transmit a single data packet is 1.152 ms. Thus total time to transmit a single packet, t_p, is the sum of the preamble and actual data packet durations.

$$t_p = 1.152 * 10^{-3} + \frac{(T_{ACK} + T_{TA})(1 - d_c)}{d_c} \tag{5.5}$$

Substituting t_p from equation 5.5 in equation 5.2, we get t, the time spent by each node for receiving and/or transmitting necessary and unnecessary packets. Thus the energy costs E_2 and E_3 incurred by all nodes in the network for transmitting and receiving necessary and unnecessary packets is

$$E_2 + E_3 = t * P * N \tag{5.6}$$

where P is the transmission or receive power and N is the total number of nodes in the network. Note that for most of the currently used radio transceivers like CC2420 [138], the transmit and receive powers are almost equal. For example, for CC2420, transmit power is 52.2 mW and receive power is 56.4 mW.

$T_R - t$ is the time period spent by nodes in not receiving or sending packets. During this time it incurs idle listening energy cost due to periodic sampling of the wireless medium. The idle listening energy cost E_1 for N nodes is given by

Table 5.4
Parameter values used for analysis, based on CC2420 datasheet

Parameter	Value
Data rate	250 Kbps (for IEEE 802.15.4 radio)
T_{TA}	192 us
T_{ACK}	352 us (Time to transmit 11 byte ACK, based on IEEE 802.15.4 standard
P	52.2 mW

$$E_1 = (T_R - t) * P * d_c \tag{5.7}$$

We conduct testbed experiments to find reprogramming time and total number of packets transmitted by all nodes in the network during reprogramming for a 5-node single-hop network of mica2 nodes. Figure 5.12-(a) and (b) show reprogramming time and number of packets transmitted, respectively, for four software change cases, one from each equivalence class: Case 1 for Class 1 (SC), Case D for Class 2 (MC), Case 7 for Class 3 (LC), and Case C for Class 4 (VLC). We see that Zephyr significantly reduces reprogramming time and number of packet transmissions. We then use the mathematical analysis presented above to find the transmission and reception energy cost ($E_2 + E_3$) and idle listening energy cost (E_1). Parameter values used for our computations are shown in table 5.4. All of these values are computed using CC2420 datasheet [138]. We use 2% duty cycle value in our calculations. The results are plotted in Figure 5.12-(c) and (d). As expected, they show that transmission/reception and idle listening energy costs are directly proportional to the total number of packets transmitted by all nodes in the network during reprogramming and reprogramming time, respectively.

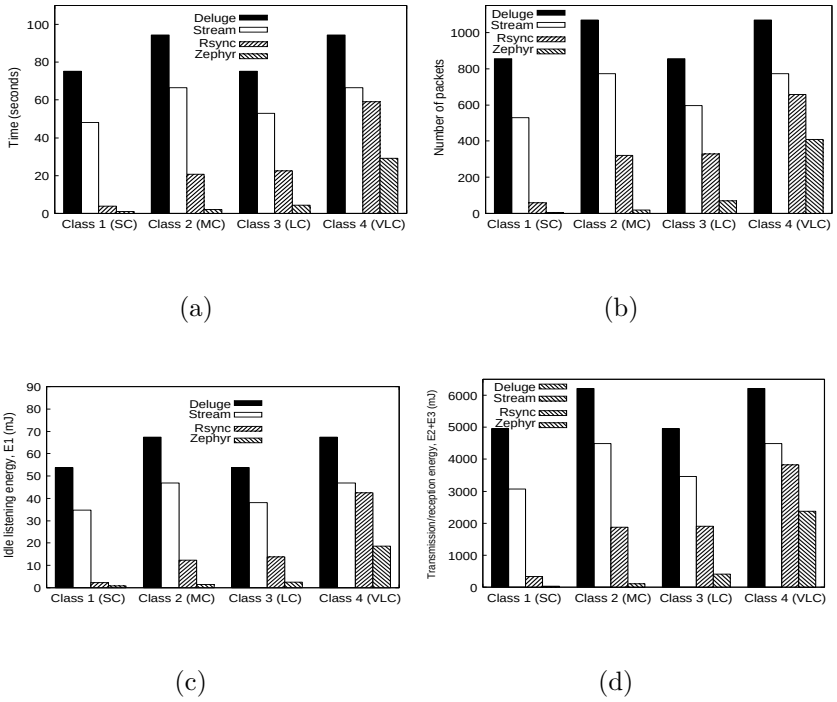


Fig. 5.12. Comparison of Zephyr with other approaches for a 5−node single hop network. (a) Reprogramming time, (b) Number of packets transmitted during reprogramming, (c) Idle Energy (E_1), and (d) Receive/Transmit Energy ($E_2 + E_3$). MAC duty cycle is 2%

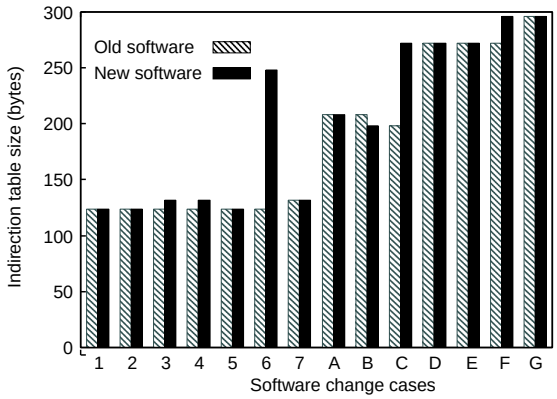


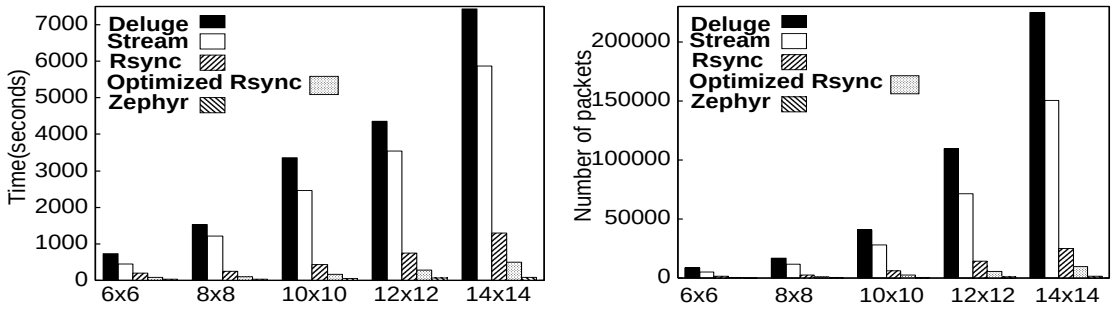
Fig. 5.13. Size of indirection table for various software change cases

5.6.4 Size of Indirection Table

Zephyr needs to allocate extra space in program memory for storing the indirection table. Figure 5.13 shows the size of the indirection table for various software change cases mentioned above. The size of the indirection table is directly proportional to the number of functions in the software.

5.6.5 Simulation Results

We perform TOSSIM [127] simulations on grid networks of varying size (up to 14x14) to demonstrate the scalability of Zephyr and to compare it with other schemes. Figure 5.14 shows the reprogramming time and number of packets transmitted during reprogramming for Case D (Class 2 (MC)). We find that Zephyr is up to 92.9, 73.4, 16.1, and 6.3 times faster than Deluge, Stream, Rsync [121], and optimized Rsync without application-level modifications, respectively. Also, Deluge, Stream, Rsync [121], and optimized Rsync transmit up to 146.4, 97.9, 16.2, and 6.4 times more packets than Zephyr, respectively. Most software changes in practice are likely to belong to this class (moderate change), and we see that application-level modifications significantly reduce the reprogramming overhead. Zephyr inherits its scalability property from Deluge since none of the changes in Zephyr (except the dynamic page



(a) (b)

Fig. 5.14. Simulation results for (a) reprogramming time and (b) number of packets transmitted during reprogramming (Case D, i.e. Class 2 (MC))

size) affects the network or is driven by the size of the network. All application-level modifications are performed on the host computer and the image rebuilding on each node does not depend upon the number of nodes in the network.

5.6.6 Best and Worst Case Scenarios

In the best case, when there is no change in the software, Zephyr needs a single COPY command in the delta script as follows:

```
COPY 0 <image_size>
```

Here 0 is the offset in the old image. It says "copy *image_size* number of bytes from old offset 0 to the new image". Note that we do not need to mention the offset in the new image as mentioned in Section 5.4.3. This delta script takes 5 bytes. However, this best case example is just hypothetical because when there is no change in the software, there is no need for (incremental) software update. So, we do not include this hypothetical case scenario in the experimental evaluation section. The small change (SC), moderate change (MC), large change (LC), and very large change (VLC) software change cases evaluated in this chapter represent the realistic spectrum

of the best case to the worst case scenarios, with SC representing the best case and VLC representing the worst case. Furthermore, we believe that most of the software change cases are of SC or MC types and thus they represent the average cases. Ideally, the worst case scenario would be the one where the two versions of the software are completely different. However, in practice, some portions of the software (e.g. driver code, core operating system code, etc) rarely change, or even if they change, they are very rarely entirely different from the previous version. So, the VLC cases presented in this chapter are good measures of practical worst case scenarios for Zephyr. In the hypothetical worst case example where the two versions of the software are completely different, Zephyr uses the following command in the delta script:

```
INSERT <software_size> <entire_software_image>
```

In this hypothetical example, the size of the delta script in Zephyr is 3 bytes more than the size of the binary image. The entire binary image is what would be sent by earlier works, such as Stream [88].

5.7 Analysis

The main idea of function call indirections is that if the positions of N_S functions have changed in the new software and those shifted functions are called C_S times in the new program code, Zephyr's technique of function call indirections causes only N_S function calls in the indirection table to be different between the two versions of the software. On the other hand, C_S function call statements are different in the baseline case. Typically $C_S >> N_S$ and thus function call indirection reduces the size of the delta script significantly. The effectiveness of function call indirection depends on the number of times the shifted functions are called in the new program.

In this section, we analyze the effect of function call indirection and the meta-commands in reducing the size of the delta script. First we define a few terms. Two function call statements in the old and new programs are said to be *identical* if they call the same function. The target addresses in the identical function call statements

may or may not be same. We use an attribute called *Preserved Similarity Index (PSI)* to quantify the amount of similarity preserved in the new version of the software with respect to the old version due to function call indirections. We define PSI as

$$PSI = \frac{C_S}{C} \tag{5.8}$$

where C_S is the number of identical function call statements in the new software that would have different target addresses than those in the old software (due to the change in the position of the corresponding functions) if function call indirections were not used. C is the total number of identical function call statements in the old and the new images. Note that $C_S \leq C$ and hence the PSI values lie between 0 and 1. A high PSI value means that the amount of similarity preserved by function call indirections is also high. For example, if PSI = 1, then without function call indirections, all the identical function call statements in the old and new images use different target addresses whereas with function call indirections, they have same target addresses (except the ones in the indirection table). Note that PSI=0 means that even without function call indirections, the identical call statements would have same target addresses because the locations of the corresponding functions are not changed by the software modification. Hence, in this case, there is no advantage due to Zephyr's function call indirections. Figure 5.15 shows the PSI values for different software change cases discussed earlier. For Case 1 and Case G where only a single parameter in the software is changed, PSI=0 as expected. Note that for most of the cases, PSI has a very high value. This suggests that most of the software change cases cause many functions to be shifted and hence without function call indirections, the number of identical function call statements with different target addresses in the two versions of the software is high. This causes the delta script to be large. This explains the observation that byte-level comparison alone is not sufficient and application-level modifications are necessary to create a small delta script.

As shown in Figure 5.16, let us consider two code segments, one in the old and one in the new new version of the software, which are *identical* except that the target addresses of n identical function call statements are different. Without function call

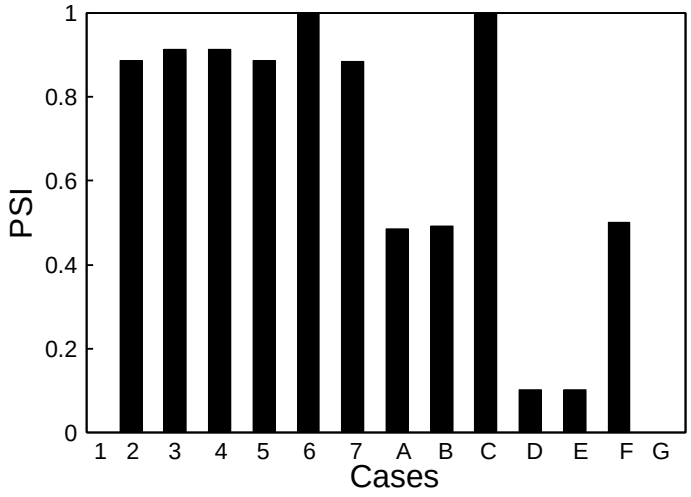


Fig. 5.15. Preserved Similarity Index (PSI) for different software change cases

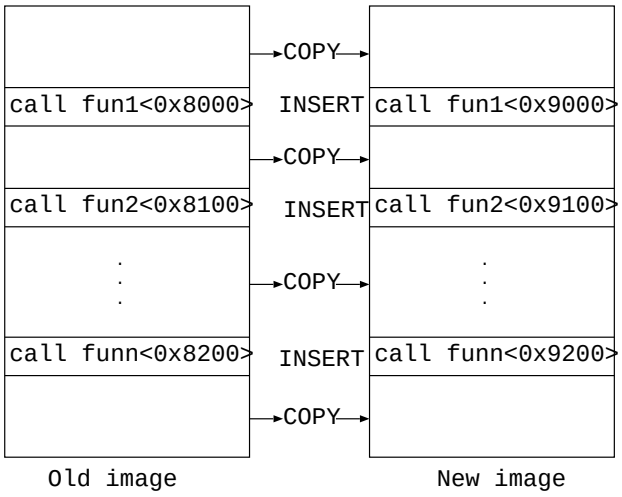


Fig. 5.16. Without function call indirections, the difference between the identical code segments require $n + 1$ COPY commands and n INSERT commands in the delta script.

indirections and metacommands, we need $(n + 1)$ COPY commands and n INSERT commands to describe the difference between these code segments. Thus the delta script for these two identical code segments is $(n + 1) * 7 + n * 7 = 14n + 7$ bytes long (since each COPY and INSERT command takes 7 bytes assuming that the target addresses of the function call statements is 2 bytes). With function call indirections,

Zephyr needs only one COPY command which requires only 5 bytes (if we do not use *newOffset* as explained in Section 6.3). However, Zephyr also needs to describe, in the delta script, the difference between the corresponding segments in the indirection table. With the REPEAT command, this can be done with $3 + 2n$ bytes. Note that the CWI command is not necessary in this context. Thus Zephyr saves $12n - 1$ bytes compared to the scheme that uses only byte-level comparison without function call indirections and metacommands. In other words, function call indirections and metacommands decrease the size of the delta script by approximately 12 times the number of times the shifted functions are called in the new program code. To evaluate the effect of function call indirections only (without metacommands), let us assume that the base line case takes 5 bytes each for COPY and INSERT commands (i.e. *newOffset* is not used). Then the delta script for these identical code segments takes $(n + 1) * 5 + n * 5$, whereas Zephyr takes 5 bytes for COPY command and $3 + 2n$ bytes for the REPEAT command. Thus function call indirections save $8n - 3$ bytes compared to the scheme that uses only byte-level comparison without function call indirections. Note that these savings are the worst case figures, since the number of function references are generally higher than the number of functions. As a result, the REPEAT command needs less than $3 + 2n$ bytes in Zephyr.

5.8 Conclusions

In this chapter, we presented a multi-hop incremental reprogramming protocol called Zephyr that minimizes the reprogramming overhead by reducing the size of the delta script that needs to be disseminated through the network. We use techniques like function call indirections to mitigate the effect of function shifts for reprogramming of sensor networks. Our scheme can be applied to systems that do not provide dynamic linking on the nodes (like the standard release of TinyOS), as well as to incrementally upload the changed modules in operating systems like SOS and Contiki that do provide the dynamic linking feature. Our experimental results show that for a large

variety of software change cases, Zephyr significantly reduces the volume of traffic that needs to be disseminated through the network compared to the existing techniques. This leads to reductions in reprogramming time and energy. As future work, we are investigating the use of multiple nodes as the source of the new code instead of a single base node to further speed up reprogramming.

6. HERMES: MITIGATING THE EFFECTS OF VARIABLE RELOCATIONS FOR INCREMENTAL REPROGRAMMING

In the previous chapter, we presented an incremental reprogramming protocol called Zephyr that transfers the difference between the old and new versions of the software, instead of transferring the entire new software. Zephyr increases the similarity between the two versions of the software by using one level of indirection for function calls to mitigate the effects of function shifts. Each function call is redirected to a fixed location in the program memory where the actual call to the function is made. In this chapter, we identify two issues with Zephyr in particular and incremental reprogramming in general — 1) Function call indirections decrease the program execution speed. Although one such indirection increases the latency of a single function call by only few clock cycles (e.g. 8 clock cycles on the AVR platform [139]), the increase in latency accumulates as the application executes repeatedly in a loop. Increase in latency in performing processing tasks means less amount of time for the sensor nodes to sleep causing the energy consumption to increase and network lifetime to decrease. 2) Function call indirections do not handle the increase in delta size due to movement of the global data variables. As the user software is changed, positions of the global variables may change and the instructions which refer to those variables may change as well between the two versions of the software. This causes a huge increase in the size of the delta. For example, for a wide range of software change cases that we experimented with, we found that the global variable shifts increase the delta size by 1369.56% on average. The increase in the size of the delta due to the relocation of the global variables depends on the number of global variables that are shifted in memory due to software modification and the number of instructions that refer to the shifted variables. From our experiments, we find that the practical

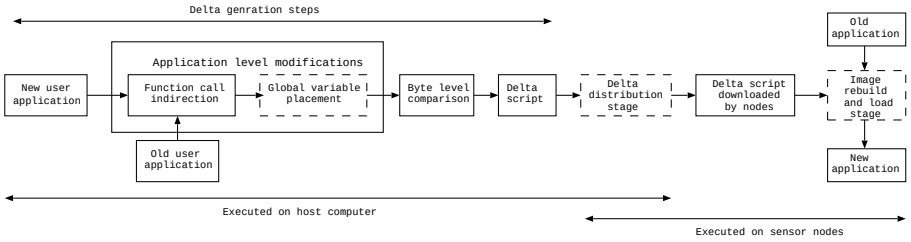


Fig. 6.1. Overview of Hermes: The stages with dashed rectangles are the ones which are introduced or modified by Hermes.

software changes generally cause many global variables to be shifted. Furthermore, compared to function shifts, global variable shifts cause the delta to be much larger because the frequency of the global variable references in the program code is generally much larger than that of function calls. This translates to proportionate increase in the time and energy required to reprogram the network. These problems exist in *all* protocols that use function call indirections and in *all* existing reprogramming protocols.

In this chapter, we present a fully functional incremental reprogramming protocol called *Hermes* (messenger of gods, in Greek mythology) which solves the problems mentioned above. It uses indirection table to mitigate the effects of function shifts and performs local optimizations at the node to avoid the latency caused by such indirection. Thus, function call indirections are used to reduce the size of the delta that is transferred wirelessly, while efficient code, without indirections, is executed, after some local transformations. Hermes also reduces the size of delta significantly by pinning down global variables to existing locations. We implement Hermes in TinyOS [14] and demonstrate it on real multi-hop testbeds as well as using simulations. Our experiments show that Deluge [92], Stream [88], protocol by Jeong and Culler [121], and Zephyr [90] need to transfer up to 201.41, 134.27, 64.75 and 62.09 times more bytes than Hermes, respectively.

6.1 Overview of Hermes

Figure 6.1 is the schematic diagram showing various stages of Hermes. As seen from the figure, the overall structure of Hermes is similar to Zephyr with some extra stages added to avoid latency due to function call indirections and eliminate the effect of global variable shifts. First, Hermes performs two application level modifications on the old and new versions of the software — one to mitigate the effect of function shifts (like in Zephyr) and the other to eliminate the effect of global variable shifts. Then the two executables are compared at the byte level using an optimized Rsync algorithm [90] to produce the delta script, which is wirelessly to all the nodes in the network using the delta distribution stage. Once the nodes download the delta script, they rebuild the new software using the old software and the received delta script. The sensor nodes run the newly rebuilt software by using bootloader to load it in the program memory. During the image load stage, Hermes avoids the funnction call indirections and consequently the latency due to such indirections. In the following sections, We describe the stages of Heremes that are different than Zephyr.

6.1.1 High-level idea of Hermes

The basic idea behind application level modifications is to mitigate the *structural changes* in the user program caused by the modification of the software so that the similarity between the old and new software is preserved and a small delta script is produced. Apart from function shifts, the other structural change caused by software modification is the global variable shifts. These result in all the instructions that refer to those variables to change between the two versions of the software. Note that local variables can also get shifted due to change in the software, but this does not cause the instructions that refer to these variables to change. To understand this, let us see how different variables are stored in RAM. As shown in Figure 6.2-a, initialized global variables are stored as .data variables in RAM followed by uninitialized global variables which are stored as .bss variables. The local variables are stored in stack

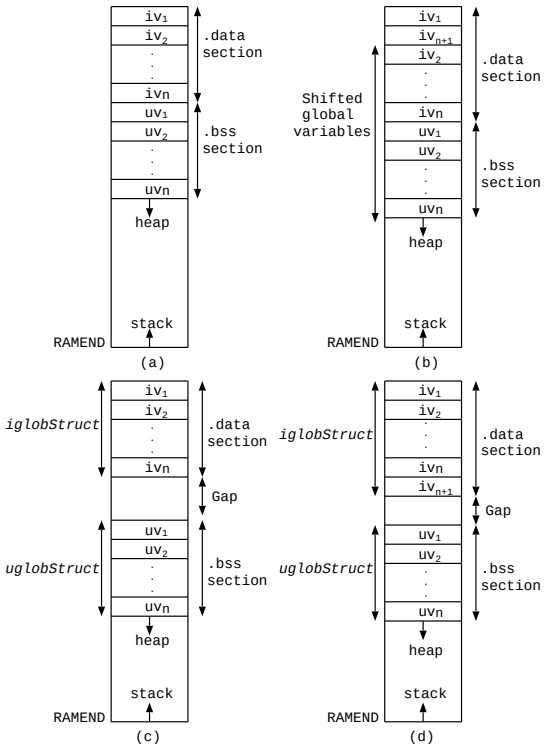


Fig. 6.2. Baseline RAM structures for (a) old and (b) new applications. RAM structures for corresponding (c) old and (d) new applications using Hermes.

which grows upward from the end of RAM. Since the local variables are referred to using the addresses relative to the stack pointer, their exact locations in RAM do not affect the size of the delta script.

To see the severeness of the global variable shifts, consider an example where a global variable is added to the Blink application. In this case, the size of the delta script produced by using only indirection table is 6090 bytes. This is disproportionately larger than the actual amount of change made in the software. The size of the delta script depends on the number of global variables that are shifted and the number of instructions that refer to those shifted variables. So, a mechanism to mitigate the effects of global variable shifts should be a very important component of application level modifications to make the delta script size proportional to the actual amount of change made in the software.

It should be noted that the actual order of the global variables in RAM is determined by the compiler implementation, not by the order in which they are declared in the user program. So the programmer has no control over the placement of the global variables in RAM. Since the location of global variables in RAM is dependent on the compiler specifics, one solution is to change the compiler itself and place the global variables such that the similarity in positions of the variables between the old and the new versions is maximized. But this calls for a complex modification to the core of a compiler, which in turn makes the solution difficult to port.

6.1.2 Placement of global variables

Since we desire a compiler-independent solution, Hermes uses the fact that members of a structure are placed in the same order in RAM as they are declared within the structure. Hermes adds one more stage (*Structure generator*) to the executable building process. If this is the first time software is being installed on the sensor nodes (i.e. no old software exists), this stage scans through the application source files and transforms the initialized global variables into members of one structure, called *iglobStruct*, and uninitialized global variables into members of another structure, called *uglobStruct*. This stage also replaces instructions that refer to the global variables by the instructions that refer to them as the corresponding members of these structures. When the software is modified, the structure generator scans through the new software to find the global variables. When such variable is found, it checks if that variable is present in the old software. If yes, it places that variable as a member of the corresponding structure (*iglobStruct* or *uglobStruct*) at the same slot in that structure as in the old software. Otherwise, it makes a decision to assign a slot in the corresponding structure for that variable (call it a *rootless* variable), but does not yet create the slot. After assigning the slots for the existing global variables, it checks if there are any empty slots in the new software. These would correspond to variables which were present in the old software, but not in the new software. If there

are empty slots, Hermes assigns those slots to the rootless variables. If there are still some rootless variables without a slot, then the corresponding structure is expanded to accommodate the rootless variables. Thus, both these structures are naturally garbage collected and the structures expand on an as-needed basis. For example, let default RAM structures for old and new applications be as shown in Figure 6.2-a and Figure 6.2-b respectively. The old application has initialized global variables $iv_1, iv_2, ..., iv_n$ in the .data section and uninitialized global variables $uv_1, uv_2, ..., uv_n$ in the .bss section. Let a single initialized global variable iv_{n+1} be added to .data section due to the modification in the software and the compiler places it after iv_1 (Figure 6.2-b). As a result, global variables $iv_2, iv_3, ..., iv_n, uv_1, uv_2, ...uv_n$ are shifted to new positions in RAM causing all the instructions in program memory that refer to these shifted variables to vary between the two versions of the application. This results in a large delta script. Hermes uses the two structures, *iglobStruct* and *uglobStruct*, to put .data and .bss variables respectively as shown in Figure 6.2-c for the old application. Hermes also leaves some space between .data and .bss sections to allow the former to grow with less chance of the latter being straddled which would cause an undesirable shift in the uninitialized global variables. In Section 6.4, we discuss how Hermes avoids this gap. In the new application (Figure 6.2-d), Hermes places the added variable iv_{n+1} at the end of the .data section so that the variables which are common between the two versions of the application are located at the same locations in RAM. So the instructions referring to the global variables that exist in both the versions do not change resulting in a small delta script.

These changes in Hermes are transparent to the user. She does not need to change the way she programs. Hermes applies these changes during the executable generation process when the user invokes program compilation.

With this approach, the size of the delta script produced by Hermes for the case where one global variable was added to Blink application is 156 bytes compared to 6090 bytes when only indirection table is used (as in Zephyr). In other words, with the addition of the structure generator to the application level modification stage, the

size of the delta script is significantly reduced making it proportional to the actual amount of change made in the software.

6.2 Image rebuild and load stage

For wirelessly distributing the delta script, Hermes uses an approach similar to that of Zephyr. After the nodes download the delta script, they rebuild the new image using the script (stored as image 1 in the external flash) and the old image (stored as image 2 in the external flash). The image rebuilder stage consists of a *delta interpreter* which interprets the COPY command by copying the specified number of bytes from the specified location in the old image to the specified location in the new image. All these locations are specified in the COPY command of the delta script. The interpreter inserts the bytes present in the INSERT command at the specified location in the new image. The new image is stored as image 3. The bootloader then loads the new software from image 3 of the external flash to the program memory (Figure 6.3). In the next round of reprogramming, image 3 becomes the old image and the newly rebuilt image is stored as image 2. Next we describe the processing at the bootloader when creating the executable image.

Avoiding latency due to indirection table: As mentioned earlier, Hermes uses Zephyr's approach of function call indirections to mitigate the effects of the function shifts. Use of one extra level of indirection increases the latency of the user program. Though it might look like one such indirection increases the time taken for one function call by only few clock cycles (e.g. 8 clock cycles for the AVR platform), it should be noted that the increase in latency accumulates over time. This is especially true for sensor networks where applications typically run in a loop — sample the sensor, process the sensed data, send data to some sink node, and then repeat the same process. Many functions are called in each iteration of the loop and the latency increases over time.

To solve this problem, we observe that there are two conflicting requirements: we need indirection table to reduce the size of the delta script and we need to remove

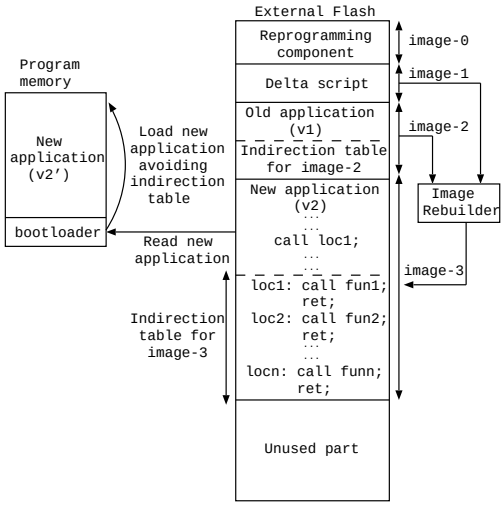


Fig. 6.3. Image rebuild and load stage. The right side shows the structure of external flash in Hermes.

any indirection for optimized execution speed. We solve this by having the sensor nodes store the application with indirection table in the external flash, but we change the bootloader to avoid using indirection table. As shown in Figure 6.3, when the bootloader loads the new image (image-3) from external flash to program memory, it eliminates the indirection by using the exact function address from the indirection table. For example, in Figure 6.3, when the bootloader reads *call loc1*, it finds from the indirection table that the actual target address for this call instruction is *fun1*. So when writing to program memory, it writes *call fun1* instead of *call loc1*. Thus as shown in Figure 6.3, the application image in program memory (v_2') is different from that in the external flash (v_2) in that it does not use indirection table. In this way, the sensor nodes still possess the program image with the indirection table in the external flash which helps to rebuild the new image in future, and yet the currently running instance of the program image does not use the indirection table and is thus optimized for execution speed. With this, we put forward a new idea for reprogramming sensor nodes — since radio transmissions are the most expensive operations, optimize for the transfer and let the sensor nodes perform some inexpensive local operations to optimize for execution speed.

6.3 Failure Handling

In this section, we discuss one of the problems associated with all incremental reprogramming approaches, which, to the best of our knowledge, has not been considered by any previous work. Let us consider a situation where a node n_1 (or a set of nodes) goes into a *disconnection state* for some time. A disconnection state means all the links of a node have failed, in a transient manner. In practical sensor network deployments, nodes may get disconnected from the rest of the network for some time due to various reasons, such as time-varying nature of wireless channels, changing environmental conditions, transient software or hardware failures, battery outages, etc.

Let $\Delta_{i,i+1}$ represent the delta script that can be used to build the application code of version v_{i+1} from the code of version v_i, i.e. $v_i + \Delta_{i,i+1} = v_{i+1}$. Before going to the disconnection state, let the node n_1 had v_j version of the application code. When n_1 comes out of disconnection state, let us suppose that it missed $\Delta_{j,j+1}, ..., \Delta_{k-1,k}$ versions of the delta script, where $k - j \geq 1$. In other words, n_1 missed one or more incremental code updates while it was in the disconnection state. After coming out of disconnection state, n_1 needs v_{k-1} version of the application code and $\Delta_{k-1,k}$ version of the delta script to build the code of version v_k ($v_{k-1} + \Delta_{k-1,k} = v_k$). It can download $\Delta_{k-1,k}$ from its neighbors. However, n_1 does not possess v_{k-1} version of the user application. Thus it cannot build the latest version of the application. Note that this is a correctness problem, not only a performance issue.

Hermes provides the following intuitive solution to this problem. Note that the actual dissemination of the delta script occurs using 3-way handshake of *advertisement-request-data* as in Deluge [92]. Nodes periodically advertise their metadata consisting of the version number of the images that they currently possess. When a node receives an advertisement message with $\Delta_{i-1,i}$ version of the delta script, it checks if it has $\Delta_{i-2,i-1}$ version of the delta script. If yes, then as usual it requests the $\Delta_{i-1,i}$ version of the delta script (i.e. image-1), downloads it, and rebuilds the new image using the

downloaded delta script and the old image (version v_{i-1}). Otherwise if it has $\Delta_{j-1,j}$ version of the delta script and $i - j > 1$, then instead of requesting the delta script, it requests the entire image of the new application. Note that this approach works well even if a cluster of nodes in a geographical vicinity misses one or more recent delta script downloads. Some of those *out-of-date* nodes may not have any *up-to-date* neighbor. But as long as at least one out-of-date node has a functioning link with at least one up-to-date node, all the out-of-date nodes will eventually get the latest version of the entire image.

Obviously downloading the entire new image is more costly than downloading just the delta script. But Hermes compromises performance for correctness. However, it should be noted that the excessive radio transmissions for downloading the entire program image are localized only in the neighborhood of the node(s) which has missed n recent code updates.

6.4 Avoiding empty space between .data and .bss sections

One drawback of the scheme outlined above is that we need to leave some empty space between .data and .bss variables in RAM to allow for .data variables to grow in future. If this space is too small, the probability of .data variables extending beyond the empty space when the software is modified becomes high, causing the .bss variables to shift. As a result, the delta script becomes large. To avoid this situation, we need to leave sufficiently large space between .data and .bss variables in RAM. But RAM is a limited resource on the sensor nodes. For example, mica2 and micaz motes have 4KB RAM. Next we explain how we solve this problem in Hermes.

One possible soultion is to leave a large space between .data and .bss sections while compiling the application on the host computer, generate the delta script on the host computer, distribute the delta script to all the sensor nodes in the network and change the bootloader running on the sensor nodes to avoid that space. When the bootloader loads the application from external flash to the program memory, it

can change the instructions that refer to .bss variables by subtracting *gapSize* from the addresses used by these instructions where *gapSize* is the size of the empty space between .data and .bss variables. Because of the complex addressing schemes on the common sensor node platforms, an algorithm with some control flow analysis is needed. Given the tight computational and memory constraints of the sensor nodes, this may not be feasible.

To solve this problem, Hermes uses two different approaches, respectively for Von-Neumann (e.g. msp430 platform [2]) and Harvard (e.g. AVR platform [139]) architectures. In Von-Neumann architecture, a single bus is used as the instruction and the data bus. Program memory (where program code is stored) and RAM (where global variables, stack and heap are stored) share the same logical address space and therefore the same mode for addressing the two kinds of memory. As a result, we can move .bss variables from RAM to program memory and avoid the space between the .data and .bss variables in RAM. We implemented this approach on TMote [3] (msp430 platform) sensor nodes. Note that program memory is larger than RAM on the sensor nodes (e.g. TMote has 10KB RAM and 48KB program memory). Reprogramming protocol that we use occupies only about 25KB of program memory and hence enough space is available for .bss variables in program memory.

In Harvard architecture, program memory and RAM lie in separate address spaces. So, if we move .bss variables to program memory, we need to change all the instructions that use data bus to refer to .bss variables with different addressing modes to use the instruction bus instead. This increases the complexity of the implementation. Furthermore, even if .bss variables are stored in program memory, we can write to those locations only from restricted areas of the program memory (e.g. bootloader section) due to memory protection. This would disallow references to the .bss variables from general-purpose user programs. Thus for Harvard architecture, when the application is compiled on the host computer, Hermes leaves a small space between the two sections in RAM. If .data section expands beyond this space, we move *only* those .bss variables which are straddled by the .data section expansion to the end of

the .bss section. For our mica2 [1] experiments, we leave an empty space of 10 bytes between .data and .bss sections. This is not a significant number because mica2 (and also micaz) nodes have 4KB RAM.

6.5 Experiments and Results

To evaluate the performance of Hermes, we considered following software change scenarios for TinyOS applications.

Case 1: Blink to Blink with a global variable added.

Case 2: Blink to CntToLeds.

Case 3: Blink to CntToLedsAndRfm.

Case 4: CntToLeds to CntToLedsAndRfm.

CntToLeds is an application that displays the lowest 3 bits of the counting sequence on the LEDs. In addition, CntToLedsAndRfm transmits the counting sequence over the radio. To evaluate the performance of Hermes with respect to natural evolution of the real world software, we considered a real world sensor network application called eStadium [130] deployed in Ross Ade football stadium at Purdue. eStadium applications provide safety and security functionality, infotainment features such as coordinated cheering contests among different parts of the stadium using the microphone data, information to fans about lines in front of concession stands, etc. We considered a subset of the changes that the software had actually gone through, during various stages of refinement of the application.

Case A: An application that samples battery voltage and temperature from MTS310 [1] sensor board to one where few functions are added to sample the photo sensor also.

Case B: We decided to use opaque boxes for the sensor nodes. So, few functions were deleted to remove the light sampling features.

Case C: In addition to temperature and battery, we added the features for sampling all the sensors on the MTS310 board except light (e.g.microphone, accelerometer, magnetometer).
Case D: Same as case C but with the addition of a feature to reduce the frequency of sampling battery voltage.
Case E: Same as case D but with the addition of a feature to filter out microphone samples (considering them as noise) if they are greater than some threshold value.

Case 1, Case D and Case E are small changes; Case 2 is a moderate change; Case A, Case B and Case 4 are large changes; Case 3 and Case C are huge changes in the software.

6.5.1 Size of delta script

Table 6.1 shows the ratios of the number of bytes required to be transmitted for reprogramming by Deluge, Stream, Rsync and Zephyr to Hermes for the software change cases mentioned above. For Deluge and Stream, the size of the information to be transmitted is the size of the binary image while for the other schemes it is the size of delta script. A small delta script translates to smaller reprogramming time and energy due to less number of packet transmissions over the network and less number of flash writes on the node. For small changes in software (like Case 1, Case D, and Case E), the incremental reprogramming protocols perform much better. Deluge, Stream, Rsync and Zephyr take up to 201, 134, 64 and 62 times more bytes than Hermes, respectively. Koshy and Pandey [122] use slop region after each function to avoid the effects of the function shifts. Hence the delta script for their best case (when none of the functions expand beyond the assigned slop regions) will be same as that of Zephyr. But even in their best case scenario, the program memory is fragmented and the ratios of Hermes to [122] would be identical to that of Hermes to Zephyr. Table 6.1 shows that [122] requires to transmit 1.79 to 62.09 times more information than Hermes for reprogramming. This huge advantage shows

Table 6.1
Comparison of number of bytes to be transmitted by various approaches

	Deluge:Hermes	Stream:Hermes	Rsync:Hermes	Zephyr:Hermes	Hermes
Case 1	148.62	84.92	63.47	39.04	156
Case 2	34.81	19.89	12.49	4.11	666
Case 3	12.37	7.66	5.64	2.73	1874
Case 4	13.41	8.3	6.14	2.95	1729
Case A	13.52	9.01	5.96	1.79	1960
Case B	15.21	10.14	6.62	1.96	1742
Case C	5.5	3.8	3.14	2.08	5223
Case D	45.65	30.43	26.02	15.51	653
Case E	201.41	134.27	64.75	62.09	148

the importance of our approach to eliminate the effects of global variable shifts. The exact amount of advantage of Hermes over Zephyr is directly proportional to the number of global variables that are shifted in Zephyr due to change in the software and the number of times those shifted variables are referred to in the program code. For example, the addition or deletion of .data variables results in more reduction in the size of the delta script by Hermes compared to Zephyr than the .bss variables. We refer to Jeong and Culler [121] as Rsync because their approach is to generate the difference using Rsync. Their approach compares the two executables without any application level modifications. The ratios of Rsync to Hermes greater than 1 show the importance of the Rsync optimization [90] and the application level modifications (both function call indirections and global variable placements). Rsync [121] approach needs to transfer 3.14 to 64.75 times more bytes than Hermes.

6.5.2 Testbed experiments

We perform testbed experiments using Mica2 [1] nodes for grid and linear topologies. For each network topology, we define neighbors of a node n_1 as those nodes which are adjacent to that node n_1 in the specific topology. For the grid network, the transmission range R_{tx} of a node satisfies $\sqrt{2}d < R_{tx} < 2d$, where d is the separation

Table 6.2
Ratio of reprogramming times of other approaches to Hermes

	Deluge:Hermes			Stream:Hermes			Rsync:Hermes			Zephyr:Hermes		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Case 1	24.77	44.66	34.24	14.12	25.96	20	10.98	19.78	15.63	7.69	16.08	11.39
Case 2	19.02	50.67	30.16	10.62	29.45	17.8	7.66	19.21	12.27	2.25	5.71	3.6
Case 3	6.14	13.48	9.8	4.77	9.15	6.37	3.37	5.57	4.56	2.06	3.56	2.8
Case 4	6.13	13.55	10.37	4.78	9.2	6.74	3.38	6.54	4.87	1.97	3.72	2.94
Case A	6.58	14.95	11.36	4.98	10.41	7.8	3.66	6.67	5.13	1.62	2.84	2.06
Case B	7.07	15.39	11.95	5.35	10.65	8.21	3.87	7.09	5.33	1.64	2.59	2.05
Case C	3.95	6.2	4.92	2.69	4.14	3.32	2.27	3.23	2.88	1.73	2.31	2.01
Case D	26.83	76.61	45.21	18.09	44.78	27.77	16.22	40.81	25.61	8.99	22.91	14.67
Case E	36.97	78.16	59.23	23.9	47.83	36.81	21.05	42.8	29.51	13.56	25.83	17.92

between the two adjacent nodes in any row or column of the grid. The linear networks have the nodes with the transmission range R_{tx} such that $d < R_{tx} < 2d$ where d is the distance between the adjacent nodes. Due to fluctuations in transmission range, occasionally a non-adjacent node will receive a packet. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped. This kind of software topology control has been used in other works also [50, 134]. For the grid network, a node situated at one corner of the grid acts as the base node while the node at one end of the line is the base node for linear networks. We provide quantitative comparison of Hermes with Deluge [92], Stream [88], Rsync (Jeong and Culler [121]) and Zephyr [90]. Note that Jeong and Culler [121] reprogram only nodes within one hop of the base node, but we used their approach on top of multi hop reprogramming protocol to provide a fair comparison. We perform these experiments for grids of size 2x2 to 4x4 and linear networks of size 2 to 10 nodes. The results presented here are the minimum, maximum and average over these grid and linear networks.

Table 6.3
Ratio of number of packets transmitted during reprogramming by other approaches to Hermes

	Deluge:Hermes			Stream:Hermes			Rsync:Hermes			Zephyr:Hermes		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Case 1	28.54	140.31	91.83	17.05	78.5	49.87	11.02	53.28	33.2	6.23	35.43	20.26
Case 2	13.84	60.72	31.73	8.42	34.84	17.45	4.75	19.27	9	3.26	11.57	5.72
Case 3	5.93	13.03	10.4	4.16	8.21	6.45	2.89	6.34	4.73	1.67	2.66	2.12
Case 4	6.2	13.26	10.11	4.04	7.84	6.27	2.6	5.96	4.59	1.77	2.53	2.12
Case A	6.34	14.79	11.56	4.51	10.7	7.88	3.03	6.64	5.11	1.86	2.28	2.02
Case B	6.37	16.53	12.41	4.53	11.46	8.46	3.03	7.71	5.49	1.85	2.26	2.01
Case C	3.94	7.6	6.17	2.84	6.02	4.4	2.49	4.74	3.68	1.56	2.85	2.3
Case D	18.87	103.12	46.34	12.64	49.1	27.7	11.63	46.74	24.91	6.94	30.27	14.63
Case E	46.67	194.19	124.29	26	114.93	76.91	20.65	87.28	59.27	12.54	53.18	35.12

Reprogramming time and energy

Time to reprogram the network is the sum of the time to download the delta script and the time to rebuild the new image. We used the approach of [50] to measure the network reprogramming time. Table 6.2 compares the ratio of reprogramming times of other approaches to Hermes. As expected, Hermes outperforms the non incremental reprogramming protocols Deluge and Stream significantly. Hermes is also 2.27 to 42.8 times faster than Rsync [121]. This illustrates that application level modifications that Hermes applies are very important in reducing the time to reprogram the networks. As mentioned above, the best case scenario for Koshy and Pandey [122] is same as that of Zephyr. Hermes is 1.62 to 25.83 times faster than Zephyr. This shows how Hermes' technique to eliminate the effects of the global variable shifts translates into speeding up the reprogramming process. To see the significance of these improvements, let us consider Case E. Deluge, Stream, Rsync, Zephyr, and Hermes took 648.68, 347.19, 299.78, 196.06, 195.06 and 14.24 seconds respectively to reprogram the 4x4 grid. Note that Hermes is most effective for small or moderate software change cases (like Case 1, Case 2, Case D and Case E) which are more likely to happen in practice. The time to rebuild the new image at the sensor node depends on the size of the delta script,

but is small compared to the total reprogramming time. In all these experiments, the image rebuild time even on the resource-constrained sensor nodes is less than 6 seconds which is small compared to the total reprogramming time (in the order of several minutes).

Among the various factors that contribute to the energy consumed during reprogramming, two important ones are the amount of radio transmissions and the number of flash writes (the downloaded delta script is written to the external flash). Since both of them are proportional to the number of packets transmitted in the network during reprogramming, we take the total number of packets transmitted by all nodes in the network as the measure of energy consumption. Table 6.3 compares the total number of packets transmitted by all nodes in the network using Hermes with other schemes for the above mentioned grid and linear networks. Like reprogramming time, Hermes reduces the number of packets transmitted during reprogramming significantly compared to other approaches. As indicated by the ratios of Zephyr to Hermes, the elimination of the global variable shifts results in a very large savings (1.56 to 53.18 times) in energy.

Execution speed

In order to demonstrate latency improvement for Hermes due to the use of the technique to avoid the indirection table, we considered a typical sensor network application which operates in a loop with each run of the loop consisting of *work* and *sleep* periods. In the work period, a node samples all the sensors on MTS310 sensor board [1], processes the sampled data and sends the data to the cluster head. In the sleep period, the node goes to sleep to save energy. All function calls happen in the work period. Figure 6.4 shows the additional latency due to indirections in all function calls during the work period. That is, the amount of time taken by Zephyr is larger than that by Hermes by the amount shown in Figure 6.4. By removing the indirection table, Hermes saves this latency, enabling lower duty cycle. So the nodes

can sleep for this extra time and hence the amount of energy saved is significant in the long run.

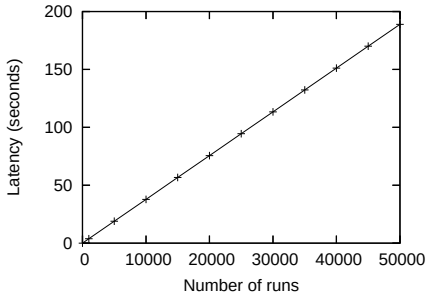


Fig. 6.4. Execution latency due to indirection table

6.5.3 Simulation Results

We perform TOSSIM [127] simulations on grid networks of varying size (up to 14x14) to demonstrate the scalability of Hermes and to compare it with other schemes. Table 6.4 shows the reprogramming time and number of packets transmitted during reprogramming for Case E. We find that Hermes is up to 94, 70, 54, 34 times faster than Deluge, Stream, Rsync and Zephyr respectively. Also, Deluge, Stream, Rsync and Zephyr transmit up to 149, 97, 74 and 46 times more number of packets than Hermes respectively. Hermes is as scalable as Deluge since none of the changes in Hermes affects the 3-way code dissemination handshake or changes with the scale of the network. All application level modifications are performed on the host computer and the image rebuilding on each node does not depend upon the number of nodes in the network. These simulation results also show that as the network grows larger, Hermes' advantage over existing protocols increases. This happens because with the increase in the network size, the existing protocols face more contention and collisions as they need to transfer more bytes than Hermes.

Table 6.4
Simulation results: Ratio of reprogramming time and number of packets transmitted by other approaches to Hermes

	6x6		8x8		10x10		12x12		14x14	
	Time	# Pkts	Time	# Pkts	Time	# Pkts	Time	# Pkts	Time	# Pkts
Deluge:Hermes	27.41	61.11	53.61	60.07	70.87	73.02	76.88	105.68	94.3	149.82
Stream:Hermes	15.55	34.16	40.01	38.68	48.25	45.4	53.28	67.66	70.52	97.55
Rsync:Hermes	12.34	26.68	2.12	27.87	28.43	34.22	38.16	49.56	54.43	74.77
Zephyr:Hermes	8.31	17.33	10.69	16.81	13.93	21.22	22.73	29.96	34.27	46.28

6.6 Analysis

A principle design technique of Hermes is to make sure that if a global variable is present in both the old and the new versions of the software, it is placed at the same location in memory. As a result, the instructions in the program code that refer to these common variables do not change between the two versions of the software. In existing incremental reprogramming approaches, these common variables may not be placed in the same memory location, causing the delta script to be large. Thus, the effectiveness of Hermes depends on the number of global variables that are shifted in the new version of the software (without the use of Hermes) and the number of times those shifted variables are referenced in the program code.

In this section, we analyze the performance of Hermes in terms of reduction in the size of the delta script by keeping the common global variables in the same memory location in the new version of the software as in the old one. First, we define a few terms. Let two instructions — one in the old and the other in the new version of the software — referencing the same global variable be called *identical references*. Note that in the baseline case, which may not place a common global variable at the same memory location, the actual variable addresses in the identical instructions may be different. But in Hermes, the variable addresses in the identical references are made identical. We use an attribute called *Preserved Similarity Index (PSI)* to quantify

the amount of similarity preserved by Hermes in the new version of the software with respect to the old version. We define PSI as

$$PSI = \frac{R_S}{R} \tag{6.1}$$

where R_S is the number of identical references in the new software that would have different variable addresses than those in the old software if Hermes were not used. R is the total number of identical references in the old and the new images. Note that $R_S \leq R$ and hence, the PSI value lies between 0 and 1. A high PSI value means that the amount of similarity preserved by Hermes is also high. For example, if PSI = 1, then without Hermes, all the identical references in the old and new images use different global variable addresses, whereas with Hermes, they have the same address. This translates to a large advantage due to the use of Hermes. Note that PSI = 0 means that even without Hermes, the identical references would have same global variable address because the memory locations of the corresponding global variables are not changed by the software modification. Hence, in this case, there is no advantage due to Hermes. Figure 6.5 shows the PSI values for different software change cases discussed earlier. Note that for many cases, PSI > 0.9. This suggests that many software change cases cause a lot of global variables to be shifted. Hence, without avoiding the effects of such shifts, the number of identical references with different global variable addresses in the two versions of the software is high. This causes the delta script to be large. This also explains the observation that byte-level comparison alone is not sufficient, and we need to enhance the structural similarity between the two versions of the software by keeping locations of global variables identical to create a small delta script.

Next we quantify the effect of global variable shift on the size of the delta script. As shown in Figure 6.6, let us consider two code segments in the old and the new versions of the software, which are *identical* except that the global variable address of n identical references are different. Without Hermes, we need $n+1$ COPY commands and n INSERT commands to describe the difference between these code segments. The delta script would therefore be $(n+1) * 7 + n + 7 = 14n + 7$ bytes long (since

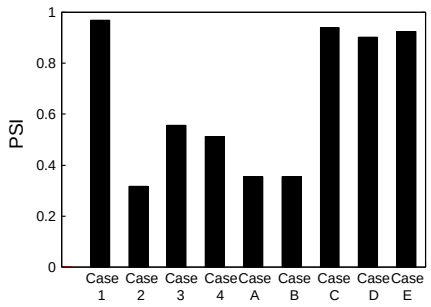


Fig. 6.5. Preserved Similarity Index (PSI) for different software change cases

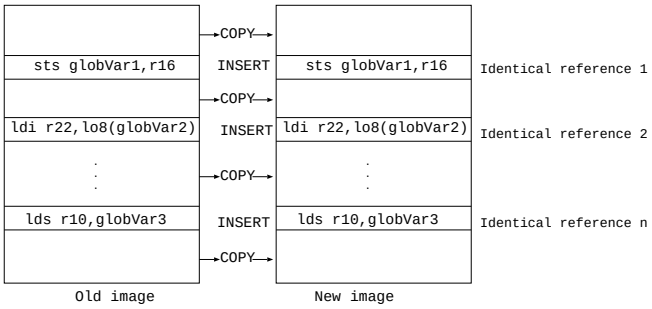


Fig. 6.6. Without Hermes, the difference between the *identical code segments* require *n+1* COPY commands and *n* INSERT commands in the delta script. Here *ldi, sts,* and *lds* are different instructions that refer to the global variables.

each COPY and INSERT command takes 7 bytes assuming that the address of a global variable is 2 bytes). With Hermes, only one COPY command is required, which is 7 bytes long. In other words, Hermes decreases the size of the delta script by (approximately) 14 times the number of times that the shifted global variables are referenced in the new program code. Note that this analysis only quantifies the reduction in the size of the delta script due to the shifted global variables.

6.7 Conclusions

In this chapter, we presented a multi-hop incremental reprogramming protocol called Hermes that minimizes the reprogramming overhead by reducing the size of

the delta script that needs to be disseminated through the network. To the best of our knowledge, we are the first ones to use techniques to mitigate the effects of global variable shifts and avoid the latency caused by function call indirections for incremental reprogramming of sensor networks. Our scheme can be applied to systems like TinyOS which do not provide dynamic linking on the nodes as well as to incrementally upload the changed modules in operating systems like SOS and Contiki that provide the dynamic linking feature. Our experimental results show that for a large variety of software change cases, Hermes significantly reduces the volume of traffic that needs to be disseminated through the network compared to the existing techniques. This leads to reductions in reprogramming time and energy.

7. VARUNA: FIXED COST MAINTENANCE IN STEADY STATE

Wireless ad-hoc and sensor networks use various dissemination protocols [62, 88, 92, 140,141] for one-to-many communication to disseminate information to all or a subset of nodes in the network. Examples of such communication are base station sending code updates for *wireless reprogramming* of the network, sending network commands or queries to nodes in the network. These dissemination protocols incur energy expenditure not only during the information dissemination phase but also during the *steady state* when no dissemination is actually being done. The need for energy expenditure in the steady state arises from dynamic changes to the network topology. Such changes are caused by the failure-prone nature of radio communications, node mobility, and incremental node deployment. For the rest of the chapter, we will use the term *steady state* to denote the state when no one-to-many information dissemination is taking place in the network, though the network will be performing other functionality, such as data collection from the many sensor nodes toward the base station.

Because of transient failures, nodes may remain disconnected from other nodes in the network for some time and may miss the information dissemination that had occurred during that period. After they come out of disconnection, they must be able to detect the data-item inconsistency and then initiate the process to become up-to-date. Inconsistency of information may also happen due to incremental node deployment or node mobility which causes a node to move into a region where its neighbors have received an update. For convenience, we are going to refer to all these events that can cause a node to get out-of-date as *topology changes*. Importantly, since these topology changes can happen at arbitrary time points and are not scheduled, any protocol to keep the network up-to-date needs to execute on a continuing basis.

Inconsistent data items can have serious consequences. For example, in wireless reprogramming, running an old version of the code could lead to wrong computation leading to erroneous aggregation and finally incorrect data being received at the base station. Even worse, different versions of the code in the network can cause the the network to be partitioned.

The traditional way of enhancing the dependability of the dissemination protocols in the presence of the unpredictable topology changes is periodic advertisements (or some variations) of some *metadata* by each node. For example, this is the approach used in the Trickle algorithm [107] which is used as the basic building block by most of the current dissemination protocols [88, 92, 102]. The metadata is a compact representation of the data-item that a node currently has. The representation has to be such that, by inspecting the metadata, a node can determine if it needs the corresponding data-item for it to become updated. A common case of metadata is a monotonically increasing version number for the data-item that the node currently has. When a node hears an advertisement from a neighbor with a newer version of the data-item than it currently has, both enter the dissemination phase through which the data-item is actually exchanged. This can be accomplished through one of several well-known protocols such as Deluge [92], Stream [88], etc.

Radio communication is often the most significant source of energy consumption in sensor networks. The problem with continuous periodic advertisements is that the steady state energy cost increases linearly with the steady state interval, which is the most dominant phase in a node's lifetime. In fact, in practice, learning *when* to disseminate a data-item can be much more costly than disseminating the data-item itself and as a result, steady state energy cost is several orders of magnitude higher than the energy cost during actual data-item dissemination phase. For example, Deluge, the default reprogramming protocol for TinyOS [14]-based sensor networks, performs periodic broadcast of the advertisement packets every 2 minutes in the steady state. Periodic advertisements at this rate for one day requires the same amount of radio transmissions as disseminating a 25 KB program code. The steady

state energy cost can be reduced by increasing the advertisement interval. However, the interval cannot be increased significantly because it increases the *detection latency*, the time taken by the nodes to determine whether they have inconsistent data-items. This in turn increases the probability of the communication between nodes with different versions of the data-item, which is a serious concern as we have seen above.

Our holy grail is to break this barrier of continuously increasing energy expenditure for state maintenance in the steady state of the network, and achieve a constant maintenance cost, independent of the duration of the steady state. We achieve this goal in the common case through our protocol called *Varuna*. Common case implies reasonable link reliabilities and reasonable memory allocation for state maintenance. To achieve this, we make two fundamental observations and leverage them.

First, if the neighborhood topology and the metadata of a node have not changed since its last advertisement transmission, then the node does not need to send any advertisement message. In periodic advertisement schemes like Trickle, practically most of the advertisements in the steady state are, therefore, unnecessary. A node can determine trivially whether its metadata has changed, through a local lookup. However, determining whether the neighborhood topology has changed is difficult and requires wireless communication among the neighboring nodes. The periodic advertisement in Trickle is essentially a way for a node to check if the neighborhood topology has changed, and if so, inform the "new" neighbors about its metadata. In Varuna, on the contrary, a node transmits advertisement messages only when required—either its metadata or local neighborhood or both have changed. Let us group all communication arising from a node into two categories—one-to-many information dissemination kind, and all the others. We will call this latter category *User Application (UA)* traffic. In Varuna, each node observes the communication pattern of UA packets of its neighbors to determine if its neighborhood has changed. Advertisement message is transmitted only when a node hears radio transmission from "new" neighbors.

The problem with the above observation is that it is impossible to determine the change in neighborhood topology based solely on communication pattern of the

neighbors. For example, application-specific decisions at a node (say, n_1) may cause it not to use a link to its neighbor (say, n_2). Such a decision can be taken for example by a routing protocol that determines there is no route to some destination or the route should not use the link n_1 to n_2. In that case, n_2 does not have a way of determining, by observing the UA packets alone, if n_1 is still a neighbor or n_2's neighborhood has changed. Therefore, we complement the first observation with a *second* one. It is critically necessary for a node to be up-to-date only when it is communicating with other nodes. This is so that stale metadata is localized to the out-of-date node only and is not propagated to other nodes. At worst, the out-of-date node may have to discard the results of some local computation that it might have performed while it was out-of-date. A fundamental problem in the premise of the previous approaches is that *every* node is responsible for ensuring that all nodes in the network are up-to-date. This means that each node needs to advertise periodically in the steady state to bring typically a few, if any, nodes up-to-date. Instead, Varuna shifts this responsibility to those few nodes which *think* that they may be out of date. Part of the design complexity of Varuna lies in solving the following question: how does a node determine that it may be out-of-date, without generating continuous transmissions for state maintenance?

In Varuna, we tradeoff some state maintenance for removing the cost of continuous advertisements by having each node maintain a small amount of state (of the order of 100 bytes) through which each node keeps track of which nodes it has heard UA packets from, since it last got updated. Also, we relax the logical invariant that if a node n_1 communicates with a node n_2, the information at the two nodes will be made consistent (and updated to that of the more up-to-date node). Let us say n_1 is out-of-date and n_2 is up-to-date. Then, in Varuna, n_2 may send a UA packet to n_1 without them realizing the inconsistency, but whenever n_1 communicates with n_2, the two nodes will detect the inconsistency and switch to the information dissemination phase. This notion of *eventual consistency* may be sufficient in many application

contexts because it prevents stale information from the out-of-date node propagating through the network.

Our contributions in the chapter are:

1. We present the first protocol for maintenance of up-to-date information in a multi-hop wireless network that does not incur a monotonically increasing cost (in terms of energy) with the length of the steady state period.

2. We show how a reasonable amount of local state maintenance can avoid the energy cost of transmissions to determine when a node is (possibly) out-of-date.

3. Our experimental and simulation results show that the actual gains realized over the current state-of-the-art is two orders of magnitude, for a steady state duration of just few days. This benefit grows linearly with increasing duration.

7.1 Trickle Overview and Problems

Trickle is the standard steady state algorithm for one-to-many information dissemination in sensor networks [88, 92, 102, 141] and forms our reference comparison point. In Trickle, each node broadcasts its advertisement message once every time interval randomly chosen from $[\tau/2, \tau]$ if it has not heard more than k identical advertisements in that interval. An advertisement contains the metadata about the data-item the node has. The metadata in this context is the version number of the data-item. When a node hears an advertisement with different metadata than its own, it sets $\tau = \tau_l$. When it hears advertisement with same metadata as its own, it keeps on doubling τ in the successive intervals. A protocol like Deluge [92] which uses Trickle for code dissemination stops this increment after reaching some threshold, $\tau = \tau_h$. The suppression of advertisement broadcast (if a node has heard more than k identical advertisements in an interval) is necessary to ensure that redundant advertisements are not broadcast and it is scalable with high node density. Clearly, without loss, collision, and with perfect time synchronization of the interval τ among

the sensor nodes, the number of advertisement broadcasts in any time interval within a single hop is bounded by k. The authors of Trickle show that with loss, collision, and lack of synchronization, the number of advertisement broadcasts in a single period τ is $O(logN)$ where N is the number of nodes within a single hop. However the number of advertisements in a given period $T(>> \tau)$ is $O(T)$. This linear increase in maintenance cost with time results in continuous energy drain in the steady state.

The steady state energy cost can be reduced by increasing the advertisement period. However, this has several problems. First, the increase in the advertisement period also increases the detection latency, the time taken by a node to realize that it is out-of-date. Detection latency and steady state energy cost have an inverse relationship—a smaller advertisement period decreases the detection latency but increases the maintenance cost and vice-versa. Trickle handles this tradeoff by decreasing the advertisement period when data-item inconsistency is detected and increasing it when nodes are up-to-date. Thus, Trickle decreases the propagation time during the dissemination phase and reduces the maintenance cost during the steady state. To ensure acceptable detection latency, advertisement interval cannot be increased arbitrarily. In Deluge, the maximum interval is 2 minutes by default. Second, although increasing the advertisement period reduces the energy cost, this is only a constant order improvement. The steady state cost still increases linearly with the steady state duration. Since a sensor node spends most of its time in the steady state, the linearly increasing energy cost becomes a serious problem in the long run.

Third, if advertisement interval is greater than the time required by a node to download the code, this makes it possible to have communication between the nodes with different versions of the code. As shown in Figure 7.1, let us suppose a node n_1 goes into a transient *disconnection state* during the time interval $[t_1, t_2]$ during which it misses a code update. We define a node to be in disconnection state if it has no functioning incoming and outgoing link. Let its neighbor n_0 download the new code during this interval. Since the advertisement interval is large, n_0 and n_1 may exchange UA packets before the next advertisement, i.e. before they detect the code

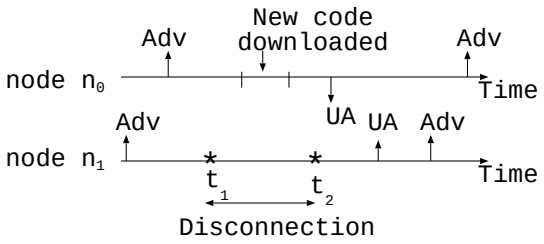


Fig. 7.1. If advertisement interval is greater than code download time, inconsistent nodes may communicate, possibly resulting in undesirable network behavior.

inconsistency. Note that this may have serious effects—the network may trigger a false alarm, or even worse, it may not detect critical events like wildfire. To avoid this problem, the advertisement interval must be less than the code download time of a node. Code download time of a node is generally in the order of few minutes. Thus the advertisement interval cannot be made arbitrarily large, which increases the steady state energy cost. In this discussion, we have used Trickle as an example. *All one-to-many information dissemination protocols in wireless networks today suffer from the problem mentioned above—monotonically increasing energy cost with the duration of the steady state.*

7.2 Design Background

Without loss of generality, we present Varuna with respect to wireless code dissemination in sensor networks. Varuna, however, is applicable to a broad range of one-to-many dissemination protocols for wireless ad-hoc networks. We first explore several intuitively appealing approaches that can reduce the maintenance cost for the steady state interval and point out the flaw that besets each approach.

7.2.1 Piggybacking metadata in UA packets

Instead of periodically advertising the metadata, a node can piggyback the metadata in each UA packet transmission because the energy cost of piggybacking is significantly lower than transmitting a separate advertisement packet. However, since metadata can be quite large, piggybacking reduces the number of bytes available in a packet for the application. For example, in Deluge, for each user application, the metadata is 12 bytes and, where dissemination of multiple applications is attempted, the metadata is even larger. Instead of piggybacking the entire metadata, only a hash value of the metadata can be piggybacked. However, this is again a constant order improvement and the overhead energy cost due to piggybacking in *each* UA packet transmission increases linearly with the steady-state time like in Trickle.

7.2.2 Checking neighborhood periodically

As mentioned above, if a node's metadata and neighborhood have not changed since its last advertisement transmission, it does not need to advertise its metadata. A node can check if its metadata has changed using local information, without communicating with its neighbors. However, for verifying if its neighborhood has changed, instead of energy-intensive proactive verification by broadcasting advertisement messages periodically as in Trickle, a node can simply listen for already existing UA packet transmissions from its neighbors. It is generally impossible for a node to derive the information about the change in neighborhood using the traffic pattern from its neighbors. Various application-specific decisions may cause a node not to use a particular link, making it impossible for its neighbor to know if its neighborhood has changed. However, it is practically sufficient for a node to verify the freshness of its metadata, *not with all nodes in its neighborhood, but only with the node from which it receives a UA packet*. When the nodes are not communicating, the consequence of not being up-to-date is localized to the out-of-data node only and is thus not as serious.

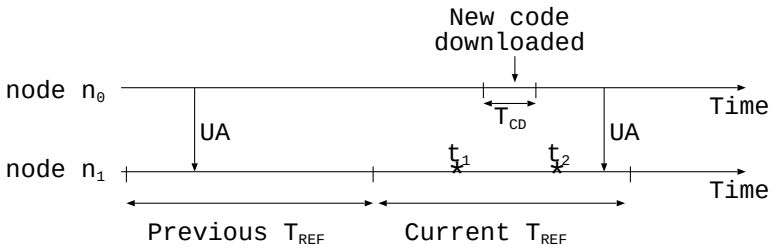


Fig. 7.2. Correctness issue if $T_{REF} > T_{CD}$.

An intuitive scheme would be for each node to maintain a neighbor table consisting of ids of the nodes from which it has heard UA packets in the last threshold time duration, call it the refresh interval, T_{REF}. In the next T_{REF} interval, if a node n_1 receives a UA packet from a node n_2 which does not exist in its neighbor table, n_1 and n_2 exchange advertisements through which they determine if they are up-to-date with respect to each other. If they are, then n_1 accepts the UA packet from n_2. Otherwise, n_1 and n_2 enter the dissemination state through which their information is made consistent. If n_1 cannot exchange the metadata with n_2 after a set number of attempts (due to link failures), it discards the UA packet from n_2 and goes back to the steady state.

Note that in this scheme each node essentially checks if its neighborhood has changed since the last T_{REF} interval using the already existing UA packet transmissions of the neighboring nodes. This scheme significantly reduces the steady state energy cost because a node needs to advertise its metadata only when its neighbor table—a measure of neighborhood topology—changes. As long as the neighbor table does not change and the node has itself not received an update, advertisements are not transmitted.

The problem with this scheme is the difficulty in choosing T_{REF} properly. It should be sufficiently large so that with a high likelihood, a node hears UA packets from all its neighbors within each T_{REF}. Otherwise, the node will needlessly perform the advertisement exchange, only to realize that both were up-to-date. In the extreme case when the neighbor table changes every T_{REF} interval, this scheme is equivalent to

Trickle with advertisement period equal to T_{REF}. However, if T_{REF} is larger than the code download time T_{CD} of a node, this scheme fails. Figure 7.2 illustrates this. Here a node n_1 goes into the disconnection state for time interval $[t_1, t_2]$ during which its neighbor n_0 downloads the new version of the code, which n_1 misses. n_1 receives a UA packet from n_0 in the previous T_{REF} interval and since $T_{REF} > T_{CD}$, let us suppose n_1 receives a UA packet from n_0 in the current T_{REF} interval also. As a result, n_1 thinks that it is up-to-date with respect to n_0 since its neighbor table has not changed. The code inconsistency, thus, goes undetected for a potentially unbounded period of time.

7.2.3 Informing neighbors of code downloads

The above correctness problem can be solved by having each node piggyback a *Code Downloaded (CD)* bit in each UA packet transmission for T_{REF} interval, each time after downloading the new version of the code. For example, in Figure 7.2, when n_1 comes out of disconnection, if it hears a UA packet from n_0 in the current T_{REF} interval, it will have the CD bit turned on. Then n_1 realizes that it has not downloaded the new version of the code in the last T_{REF} interval, but n_0 has. Thus the code inconsistency is detected.

However, even with this revised scheme, T_{REF} cannot be made larger than T_{REP}, the minimum time interval between two consecutive reprogramming procedures. Figure 7.3 illustrates this. Here n_0 and n_1 download version v_n of the code in the current T_{REF} interval. But after that, n_1 goes into the disconnection state for interval $[t_1, t_2]$. During this time, n_0 downloads v_{n+1} version of the code, which n_1 misses. When n_1 comes out of disconnection, it receives a UA packet from n_0 with CD bit turned on and believes that it is up-to-date with n_0 because n_1 also downloaded code in the current T_{REF} interval. Hence the code inconsistency goes undetected again for a potentially unbounded time.

Note that instead of piggybacking the CD bit, if n_0 had piggybacked "the number of times the code was downloaded in the current T_{REF}" or "the latest version of

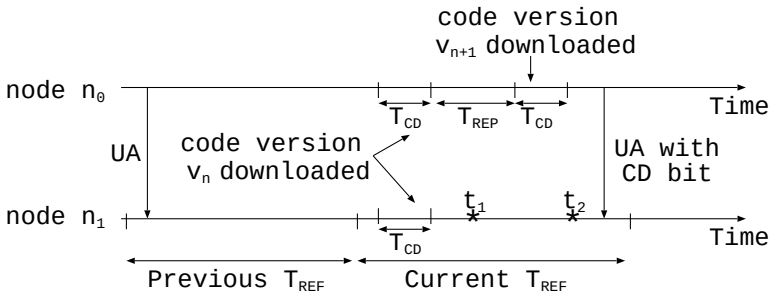


Fig. 7.3. T_{REF}, the refresh interval cannot made larger than T_{REP}, the minimum time between two successive code downloads, for correctness reasons.

the code downloaded in T_{REF}", n_1 would have detected the inconsistency. But this is generally not possible since a sensor node may be running multiple applications and thus it would need to explicitly say *which versions* of *which applications* were downloaded in the last T_{REF} interval. This information is too large to be piggybacked in every UA packet for a "large" T_{REF} interval.

Any scheme that uses threshold (refresh) time intervals to check if the neighborhood has changed between such intervals has a fundamental performance problem—since UA can be arbitrary, no matter how large a T_{REF} is chosen, a neighbor can be such that it sends UA packet at every other T_{REF} interval, causing the neighbor table to change in every T_{REF} interval. As a result, the node needs to advertise in every T_{REF} interval and thus, in terms of energy cost performance, such scheme is equivalent to Trickle with advertisement period equal to T_{REF}.

7.3 Varuna Design

Based on the above observation that a node cannot determine if its neighborhood topology has changed by monitoring the communication pattern for a finite time duration, we arrive at a very simple maintenance algorithm, called Varuna, that does not use the notion of refresh interval. The above approaches (Sections 7.2.2 and 7.2.3) try to make sure that the code inconsistency is detected when nodes with different

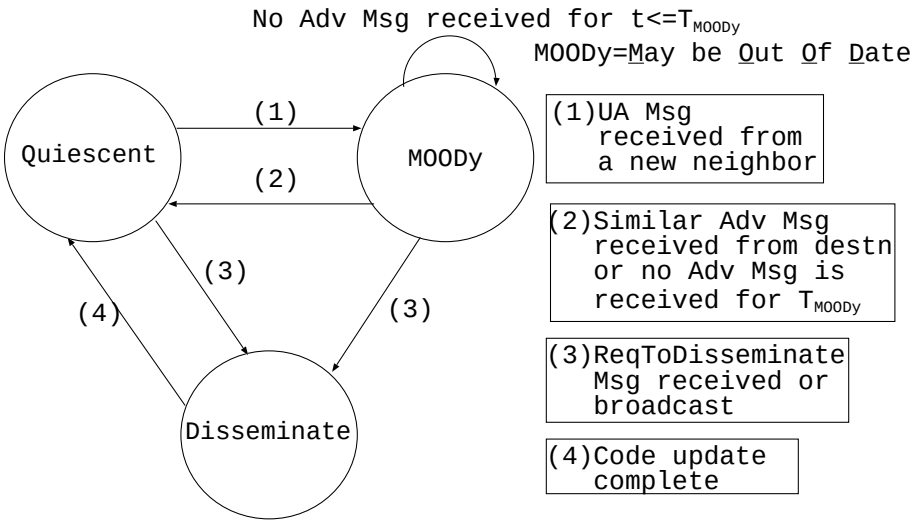


Fig. 7.4. State transition diagram of Varuna

versions of the code communicate. Varuna relaxes this requirement and, instead, guarantees that the following invariant is satisfied—*When a node receives a packet from another node with a lower version of the metadata than its own, the metadata inconsistency is detected by the receiving node.* This invariant also implies that Varuna achieves eventual consistency—even though a node n_1 may not detect inconsistency while it is receiving packets from a node n_2 which has a higher version of the metadata than n_1, eventually when n_2 receives the packet from n_1, the inconsistency is detected. We believe this relaxed form of consistency is satisfactory in most application contexts and is necessary in practice to achieve a constant cost of state maintenance. With Varuna's invariant, information does not flow from out-of-date nodes to up-to-date nodes, and thus, the erroneous result is not propagated in the network. Note that base stations (or nodes close to them) can be assumed to be always up-to-date. Thus, erroneous results from out-of-date nodes will not be collected by the base station. Before presenting a formal description, we first present an overview of Varuna. Figure 7.4 shows the state transition diagram of Varuna.

7.3.1 Design Overview

Each node maintains a neighbor table consisting of the ids of the nodes from which it has received any packet since the last time it updated its metadata (i.e. downloaded the new version of the code). When a node is booted up or its metadata is updated, the neighbor table is cleared and the node goes to the Quiescent state. When a node n_1 receives a UA packet from a node n_2, it checks if n_2 exists in its neighbor table. The case of overflow of the neighbor table is discussed later. If n_2 exists in the table, n_1 accepts the packet. If it does not, n_1 suspects that it (or n_2) **M**ay be **O**ut **O**f **D**ate (MOODy) and goes to the MOODy state where it tries to verify if n_2 has the same version of the metadata as that of n_1. We will shortly explain how this is done. If n_1 finds that both have same version of the metadata, it inserts n_2 in its neighbor table, goes to the Quiescent state, and accepts the packet from n_2. Otherwise, it goes to the Disseminate state. In the Disseminate state, the out-of-date node receives the latest version of the code from the up-to-date node(s) following any one of the available dissemination protocols [88, 92]. Varuna's design is orthogonal to that of the dissemination protocol and it can work with any of them.

Verifying if metadata is up-to-date: In the MOODy state, n_1 broadcasts *Advertisement* packet containing its metadata, source id, and a field called *dest* set to n_2. Let v_i^k $(i = 1, 2, ..., n)$ represent the version numbers of n application codes (or data-items) present in node k. When n_2 receives the Advertisement packet with *dest* set to its node id, it compares the received metadata with its own. If n_2 finds that it needs an update (i.e. $v_i^{n_2} < v_i^{n_1}$ for any i), it broadcasts a *ReqToDisseminate* packet and transitions to the Disseminate state. When n_1 and other neighbors of n_2 receive the *ReqToDisseminate* packet, they also go to the Disseminate state. If n_2 finds that it does not need an update (i.e. $v_i^{n_2} \geq v_i^{n_1}$ for all i), it broadcasts an Advertisement packet with *dest* set to NULL. When n_1 receives this advertisement packet, it verifies if it needs an update. If it does not (i.e. $v_i^{n_1} = v_i^{n_2}$ for all i), it goes back to the Quiescent state, adds n_2 to its neighbor table, and accepts the packet received from n_2.

If n_1 needs an update (i.e. $v_i^{n_1} < v_i^{n_2}$ for any i), it broadcasts a *ReqToDisseminate* packet. Neighbors of n_1 (including n_2) go to the Disseminate state after receiving this packet. Advertisement and *ReqToDisseminate* messages are transmitted using random backoff intervals—[0,ADV_RAND] and [0,DISS_RAND] respectively, to avoid collisions due to concurrent transmissions from nearby nodes.

The use of *dest* field in the broadcast Advertisement message avoids the transmission of redundant advertisements in the neighborhood of the MOODy node n_1. A node which receives the advertisement message with *dest* set to NULL or *dest* not set to its own id will reply with its own advertisement message only if required (i.e. either the receiver or the sender needs an update) and *suppression threshold* has not been reached. For example, when neighbors of MOODy node n_1 other than n_2 hear advertisement from n_1 with *dest* set to n_2, they do not advertise if they don't have to—i.e. if $v_i^n = v_i^{n_1}$ for all i, where $n \in N(n_1)$ and $N(n_1)$ is the set of neighbors of n_1 except n_2. If any node in $N(n_1)$ needs an update (i.e. $v_i^n < v_i^{n_1}$ for any i), it broadcasts *ReqToDisseminate* and goes to the Disseminate state. Otherwise, if a node n in $N(n_1)$ finds that n_1 needs an update (i.e. $v_i^n > v_i^{n_1}$ for any i) , n broadcasts its advertisement message if it has not heard more than k (suppression threshold) advertisements with same metadata as its own since it heard the advertisement from n_1. Because of this advertisement suppression, which is borrowed from Trickle, Varuna scales well with varying node density. When n_2 receives advertisement from n_1 with *dest* set to n_2, it replies by broadcasting its Advertisement message with *dest* set to NULL, irrespective of whether it is up-to-date with n_1. This is because n_1 wants confirmation about the freshness of its metadata with respect to n_2 that caused n_1 to become MOODy.

Similarly, when n_2 replies with its metadata broadcast with *dest* set to NULL, neighbors of n_2 will not broadcast advertisement if their metadata is same as n_2's. If they do not match, the neighbor node behaves similarly to the behavior when it heard the advertisement from n_1 with *dest*=n_2. In this way, even if n_1 and n_2 have the same but outdated versions of the data-item, their neighbors help them detect

the inconsistency and make them transition to the Disseminate state where they can be updated.

Retries to deal with link failures: In the MOODy state, n_1 may not receive any response to its Advertisement message from n_2 even though n_1 had received a UA packet from n_2 that triggered n_1 to be MOODy. The link between n_1 and n_2 may be functional in only one direction (n_2 to n_1), n_2 or n_1 may have moved after n_2's UA packet is received by n_1, or n_2 may have had a transient node failure. If n_1 does not receive any response, it re-broadcasts the Advertisement message after every τ interval for T_{MOODy} duration. If no response is received during T_{MOODY}, n_1 returns to the Quiescent state and discards the UA packet received from n_2.

Dealing with a full neighbor table: A node inserts a new neighbor in its neighbor table in the next available slot as long as the neighbor table is not full. When the table is full, it replaces the least recently used (LRU) neighbor with the new neighbor. The LRU node is the one from which it has not received any packet for the longest duration. Thus, in addition to the neighbor id, a neighbor table entry must contain the last time the node received a UA packet from this neighbor. An important design point of Varuna is that there is no notion of refresh time interval for clearing off the local state. Rather, Varuna uses a neighbor table which is cleared in its entirety when the node receives a code update or it is turned on.

7.3.2 Formal Protocol Description

Here we describe the local rules followed by each node in Quiescent and MOODy states. In the Disseminate state, nodes follow any of the current protocols used for dissemination [88, 92, 102]. Conceptually, the gain due to Varuna arises from the fact that a node spends most of its time in the Quiescent state, where it does not transmit any advertisement packet. It transitions to the MOODy state only when there is some likelihood that neighborhood topology has changed and therefore it is worthwhile for

the node to check if it needs to be updated. Varuna intelligently controls when the transitions to the more expensive MOODy state need to happen.

Quiescent State

A node in the Quiescent state follows these local rules.

Q.1: When a node goes to the Quiescent state upon booting up or updating its metadata, it clears its neighbor table.

Q.2: When a node n_1 receives a UA packet from a node n_2, n_1 checks if n_2 exists in its neighbor table. If it exists, n_1 accepts the packet. Otherwise, n_1 goes to the MOODy state to verify if n_2 is up-to-date with n_1.

Q.3: If a node hears an Advertisement packet from a neighbor and finds that it is up-to-date with the neighbor, the neighbor is added to the neighbor table if it does not already exist in the table.

Q.4: If a node n_2 receives an Advertisement packet from a node n_1 with *dest* set to n_2, it compares the received metadata with its own. If n_2 needs an update (i.e. $v_i^{n_2} < v_i^{n_1}$ for any i), n_2 broadcasts *ReqToDisseminate* packet, after a time interval randomly chosen from [0, DISS_RAND], and goes to the *Disseminate* state. Otherwise, it broadcasts an Advertisement packet with *dest* set to NULL.

Q.5: If a node n_3 hears a broadcast Advertisement message from a node n_1 with *dest* set to NULL or *dest* other than n_3, it compares the received metadata with its own. If it finds that it has the same version of the metadata as the received one, it ignores the Advertisement message. If they are different and n_3 needs an update (i.e. $v_i^{n_3} < v_i^{n_1}$ for any i), n_3 broadcasts *ReqToDisseminate*, after a time interval randomly chosen from [0, DISS_RAND], and goes to the Disseminate state. Otherwise, if the metadata are different but n_1 needs an update (i.e. $v_i^{n_1} < v_i^{n_3}$ for any i), n_3 broadcasts Advertisement packet with *dest* set to NULL, after a random time from the interval [0, ADV_RAND], conditioned on advertisement suppression. Advertisement suppression

implies that if the node has heard more than k advertisements with the same metadata as its own, then it will not broadcast its advertisement message.
Q.6: If a node receives a *ReqToDisseminate* packet, it goes to the Disseminate state.

MOODy state

Let n_1 be the node which transitions to the MOODy state after receiving a UA packet message from a node n_2 that does not exist in n_1's neighbor table. The node n_1 in the MOODy state follows these rules.
M.1: As long as n_1 does not receive any Advertisement message from n_2, it broadcasts Advertisement message with *dest* set to n_2 after every τ interval, conditioned on advertisement suppression.
M.2: If n_1 does not receive any Advertisement message from n_2 for T_{MOODy}, it goes back to the Quiescent state and discards the packet received from n_2.
M.3: If n_1 receives an Advertisement message from n_2, it checks its metadata with that of n_2. If they match (i.e. $v_i^{n_1} = v_i^{n_2}$ for all i) and the neighbor table is not full, n_1 adds n_2 to its neighbor table, goes to the Quiescent state, and accepts the UA packet received from n_2. If the neighbor table is full, n_1 replaces the LRU node in its neighbor table with n_2, goes to the Quiescent state, and accepts the UA packet received from n_2. If the metadata don't match and if n_1 finds that it needs an update (i.e. $v_i^{n_1} < v_i^{n_2}$ for any i), n_1 broadcasts *ReqToDisseminate* packet, after a time interval randomly chosen from [0,DISS_RAND], and goes to the Disseminate state.
M.4 Same as Q.5.
M.5 Same as Q.6.

7.3.3 Eventual consistency

Varuna ensures that if a node receives a packet from another node with a lower version of the metadata than its own, the metadata inconsistency is detected by the receiving node. So, in Varuna, communication from a node with a higher version

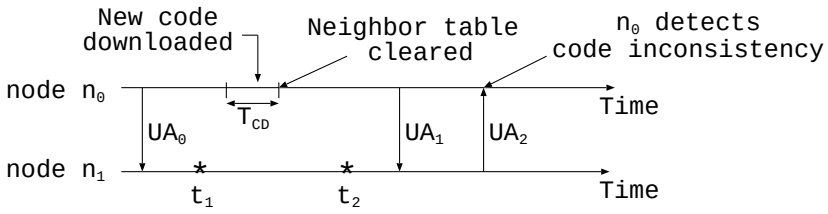


Fig. 7.5. Eventual consistency in Varuna

of the metadata to another node with a lower version of the metadata can happen, without the nodes detecting the inconsistency. For example, as shown in Figure 7.5, let n_1 go to disconnection state in the time interval $[t_1, t_2]$, during which its neighbor n_0 downloads a new version of the code. After n_1 comes out of disconnection, let it receive a UA packet, UA_1, from n_0. n_1 finds n_0 in its neighbor table since n_1 had earlier received UA_0 from n_0. Since receiving UA_0, n_1 has not cleared its neighbor table as its metadata has not changed. Thus the communication from a node with a higher version of the code (here n_0) to a node with lower version (here n_1) goes undetected. However, Varuna ensures eventual consistency—when n_0 receives UA_2 from n_1 after some time, n_0 does not find n_1 in its neighbor table as it has been cleared after downloading the code. Thus n_1 goes to the MOODy state and detects the code inconsistency. Note that in Trickle, a node with a lower version of the code can communicate with a node with a higher version of the code (and vice-versa), without them detecting the inconsistency, as illustrated in Figure 7.1.

An interesting parallel to this notion of eventual consistency is seen in data storage in massive (wireline) distributed systems. The community looks at three dimensions of the problem of data storage—data consistency, resilience to network partitions, and availability. There is convergence toward the view that to support the last two attributes, designers have to relax the consistency model to eventual consistency [142]. Likewise, here to support fixed energy cost for data-item consistency and resilience to network topology changes (due to link failures, mobility, or incremental deployment), we have to relax the consistency model to eventual consistency.

7.3.4 Fixed steady state cost

In Varuna, after a node downloads a new version of the code, it verifies its changed metadata with each of its neighbor *only once*. After this verification, if the neighbor table does not overflow, no further advertisements are necessary. So, in the common case, Varuna incurs *fixed* cost in the steady state, independent of the steady state interval. However, in some cases, a node may need to advertise occasionally due to the *poor neighbor* problem, which we define as follows. In some large networks, a node may occasionally receive a UA packet from "far neighbors" with very poor link reliabilities. Let us call such neighbors poor neighbors. This triggers the node to be MOODy. Since the MOODy node tries to verify the freshness of its metadata with a poor neighbor for a finite time duration (T_{MOODy}), the probability that it will succeed is low. As a result, the node goes back to the Quiescent state without success, and the poor neighbor is not added to the neighbor table. Every time a node receives a UA packet from the poor neighbor, though it happens rarely due to the low link reliability, it incurs the cost of transitioning to and back from the MOODy state.

The poor neighbor problem occurs very rarely because of various reasons. First, for the MOODy node to be unsuccessful in verifying the freshness of its metadata with the poor neighbor, the link reliability between them should be very poor. This means that the node will hear from the poor neighbor very rarely in the first place. Second, according to rule Q.3, whenever a node overhears an Advertisement packet with the same metadata as its own, it adds that neighbor to its neighbor table if it does not already exist in the table. This means that a node n_1 gets multiple opportunities to add a poor neighbor n_2 to its neighbor table—not only when n_1 hears directly from n_2, but also when n_2 broadcasts an advertisement in response to an advertisement message from a neighbor of n_2. This in turn means a stray message from n_2 does not necessarily cause n_1 to transition to the MOODy state and expend energy. Third, not all UA packets from poor neighbors cause n_1 to be MOODy. If the poor neighbor sends a unicast message not destined for n_1, it does not cause n_1 to be MOODy. The

one-hop unicast messages from the poor neighbor are generally not directed to n_1 as the poor node may not even know the existence of n_1 because of the poor link. Note that Varuna only tries to enforce code consistency when nodes communicate with each other. Not all communications in sensor networks are of broadcast nature. Thus the effect of poor neighbor is greatly reduced. Hence, it is only in rare cases that Varuna incurs the small increase in energy cost with the duration of the steady state period.

The advantage of Varuna over Trickle is even more pronounced in networks for rare event detection. Nodes generate packets very rarely in response to the occurrence of certain events. As a result, in Varuna, nodes need to verify their metadata very rarely. The effect of the poor neighbor problem is also significantly reduced. Trickle, on the other hand, needs to advertise periodically. The period cannot be chosen to be as large as the average event occurrence period because if events occur faster than the estimated average, the likelihood of communication between the inconsistent nodes increases, as shown in Figure 7.1. Also, if the events occur less frequently than the average estimated duration, redundant advertisements are transmitted.

7.3.5 State maintenance cost

Trickle does not require any state maintenance, while in Varuna, each node maintains state in the form of a neighbor table to reduce energy consumption due to periodic advertisements. Commercially available sensor nodes today have more memory at low cost, and this trend is likely to continue in future. Thus we believe that this tradeoff makes sense because reducing the energy consumption and increasing the network lifetime are, generally, more important than saving some memory. The neighbor table is a localized data structure and its size does not increase with the size of the network, but with the size of the neighborhood of the node. Also, as we will show from our experiments, for most practical deployments, the neighbor table consumes less than 200 bytes of memory. For “very large” and “very dense” net-

works, less than 600 bytes are enough. Consider the memory available in current sensor nodes: TelosB, micaz, IMote2, IRIS, BTNode, and SunSPOT have 10KB, 4KB, 256KB, 8KB, 180KB, 512KB RAM, respectively.

Furthermore for many sensor networks, neighbor table is a fundamental data structure. It is used by many protocols and services—MAC protocols [143, 144], AODV based routing protocols [145], 6LoWPAN [146] standard for IP based sensor networks, ZigBee [147], and many other applications. Thus, Varuna can leverage the existing structure and add to each entry the field for the time the neighbor was last heard from.

In all but some very large and very dense networks, the size of the neighbor table is small, and can be fixed beforehand. In very large and very dense networks, the size of the neighbor table can be adjusted dynamically based on runtime observations of the table overflow. Each node starts with a small neighbor table. Each overflow triggers a multiplicative increase of the neighbor table and low occupancy causes an additive decrease.

7.3.6 Detection latency

Traditionally, detection latency is defined as the time duration from the instant when a node becomes out-of-date to the instant when it *knows* that it is out-of-date. Clearly, higher advertisement rate and UA packet rate decrease detection latency in Trickle and Varuna, respectively. However, one of the basic ideas of Varuna is that nodes need not be up-to-date with *all* neighbors *all the time*. Rather, information should not flow from a lower version node to a higher version node. Thus, in this context, a more suitable definition of detection latency is the time interval from when an out-of-date node communicates with an up-to-date node for the first time to the instant when it realizes that it is out-of-date. As is obvious from Varuna's design, this detection latency is zero, because whenever an out-of-date node communicates,

its inconsistency is detected. Even with this new definition of detection latency, for Trickle this is still a function of advertisement period.

7.4 Implementation and Evaluation

We implement Varuna on TinyOS [14]. Varuna lies logically between MAC and the higher layer. MAC forwards the received UA packet to Varuna, which executes its algorithm to decide if the packet should be forwarded to the appropriate application in the higher layer. In TinyOS, *Active Message ID* uniquely identifies each *type* of packet sent from each application. For packet transmission, the higher layer application or service gives the packet directly to the MAC layer, bypassing Varuna.

In order to evaluate the performance of Varuna and compare it with Trickle, we perform testbed experiments using TelosB sensor nodes [1]. For large scale evaluation, we use TOSSIM [127] simulations. We run the network in the steady state since that is the focus of Varuna. This means in the experiments no information dissemination is taking place. We use steady state energy cost as a metric to compare Varuna and Trickle. Since the energy cost is directly proportional to the number of advertisement transmissions in the steady state, we use the total number of advertisement packets transmitted by all nodes in the network as a measure of steady state energy cost. We also quantify the memory cost for state maintenance in Varuna for various node densities and network sizes. Each entry in the neighbor table takes 6 bytes (2 bytes for neighbor-id and 4 bytes for the time when this neighbor was last heard from).

7.4.1 Testbed Results

We perform testbed experiments using a 30-node network of TelosB nodes arranged in a 5X6 grid. We vary the distance d between the successive nodes in any row or column to evaluate the performance of Trickle and Varuna with respect to node density. The output transmission power of each node is set to the minimum possible value. In Varuna, each node broadcasts UA packets after every T_{UA} interval, which is

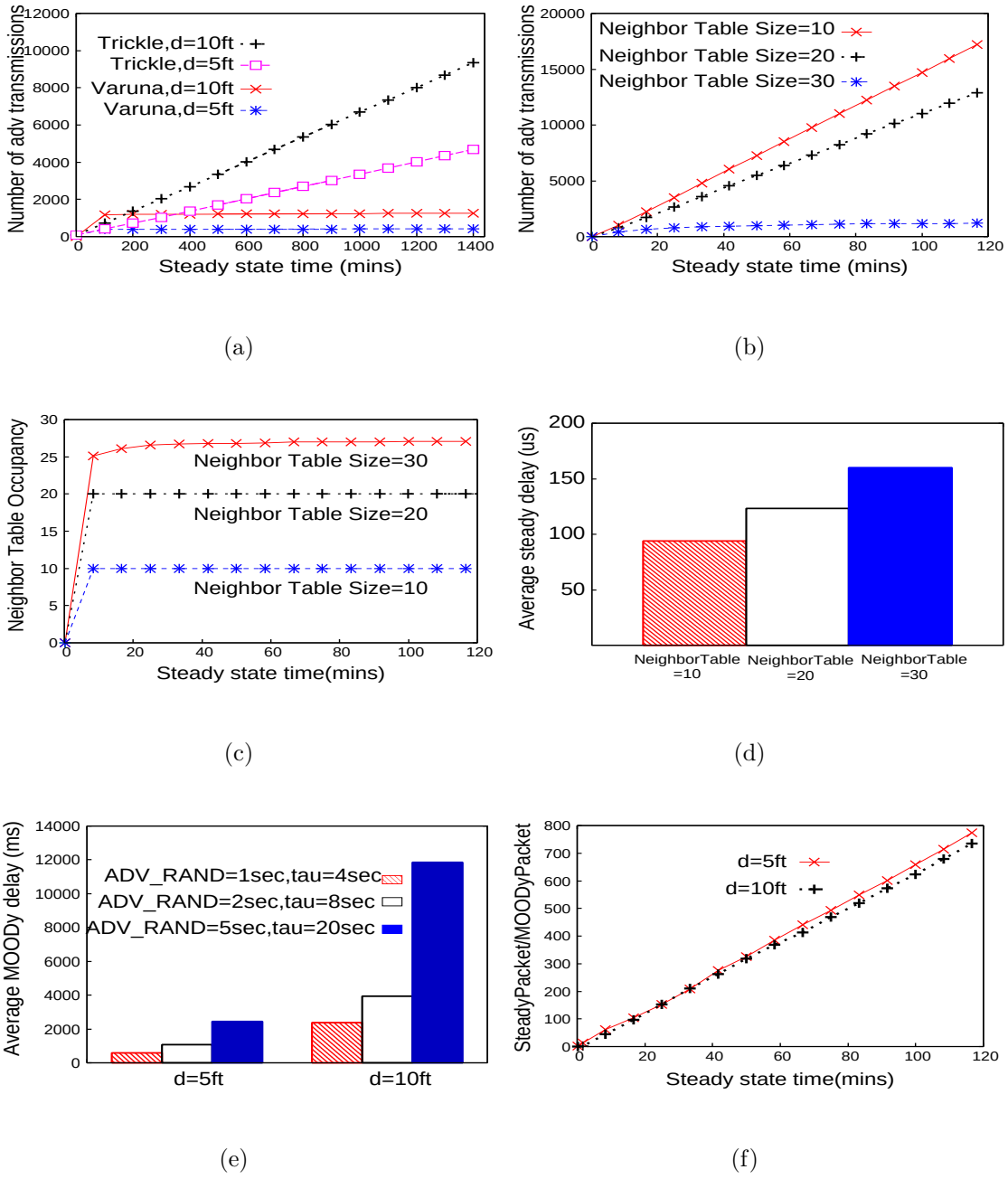


Fig. 7.6. Testbed results: Steady state energy cost as a function of time for (a) neighbor table size=30, and different grid spacings d, and (b) d=10ft and different neighbor table sizes; (c) Neighbor table occupancy vs time for d =10ft; (d) Steady delay and (e) MOODy delay; (f) Ratio of number of packets that face smaller steady delay to those that face larger MOODy delay.

Table 7.1
Parameters for the experiment.

T_{UA}	U[0,60sec]	(τ_l, τ_h)	(2sec,2min)
τ	4,8,and 20sec	k	2
ADV_RAND	1,2,and 5sec	T_{MOODy}	1min

uniformly distributed between 0 and 60 seconds. The values of the parameters used in the testbed experiments are shown in Table 7.1.

Comparison of steady state energy cost

Figure 7.6-a compares the number of advertisement transmissions by Trickle and Varuna as a function of steady state time. In this experiment, each node allocates a neighbor table of size 30, i.e. 180 bytes. As expected, the steady state energy cost increases linearly with time in Trickle. However, in Varuna, it does not increase after some time because once a node verifies the freshness of its metadata with each of its neighbors, it does not need to transition to the MOODy state and advertise anymore. In just 1 day, the steady state energy cost in Trickle is about 8 and 11 times more than that of Varuna for node spacing, $d = 10$ ft and $d = 5$ ft, respectively. As sensor networks are generally expected to operate for months, if not years, Varuna would achieve energy savings of several orders of magnitude compared to Trickle. A simple extrapolation of the data in Figure 7.6-a shows that in one month, Trickle consumes about 223 and 336 times more energy than Varuna, for $d = 10$ ft and 5 ft, respectively. Note that in our experiments nodes are not duty-cycled. With duty cycling, identical energy savings due to Varuna will take longer to be realized, increasing in inverse proportion to the duty cycle.

When the distance d between successive nodes in the grid is increased, the energy consumption increases both in Trickle and Varuna. The increase in d (equivalently, decrease in node density) causes the link quality to be poor between the neighboring nodes. As a result, in Trickle, a node may not hear k or more identical advertisement

broadcasts in its neighborhood, even though that many may have been broadcast in reality. Consequently, the node will not suppress its own advertisement. This leads to more redundant advertisements for d=10ft than d=5ft. In Varuna, poor link qualities cause many retransmissions of advertisement packets in the MOODy state and thus the energy cost is higher for the sparser network.

Effect of neighbor table size

In the above experiment, Varuna incurs fixed cost in the steady state because the neighbor table does not overflow as it is sufficiently large. If the neighbor table is small, a node n_1 may need to evict an LRU node n_2 from the neighbor table to accommodate a "new" neighbor n_3. Later, when n_1 receives a UA packet from n_2 , n_1 goes to the MOODy state (even though n_2 is up-to-date with n_1) as n_2 does not exist in its neighbor table. So, the steady state energy cost cannot be a fixed value if the neighbor table is small. Figure 7.6-b shows the steady state energy cost in Varuna for different neighbor table sizes for d =10 ft. When the size of the neighbor table is 30, the steady state energy cost is fixed. But when the neighbor table is 10 or 20, it increases linearly with time. Although not shown in the figure, obviously the slope of this linear relationship increases with the increase in the UA packet reception rate because UA packet reception causes the node to be MOODy if the source of the UA packet is not in the neighbor table. For small or moderate size networks, the neighbor table can be made sufficiently large to ensure fixed energy cost in the steady state. In the next section, we will show through simulation results that, even for very large and dense networks, the memory requirement for state maintenance is reasonable. It is fundamentally because the neighbor table size does not grow with the network size. Rather, it grows with increasing number of nodes in a neighborhood, or equivalently, the network density.

Figure 7.6-c shows how the actual neighbor table occupancy varies with time. The neighbor table occupancy shown in this graph is an average over all nodes in

the network. When 30 slots are allocated for the neighbor table, on average each node uses about 26 slots, but with 10 or 20 slots, the occupancy reaches the capacity and there is overflow. Thus, 10 or 20 slots are not sufficient, which causes the linear increase in steady state energy consumption shown in Figure 7.6-b.

Delay introduced by Varuna

Varuna intercepts the UA packet from MAC layer en route to the application, executes its algorithm to determine if the source of the packet is up-to-date with it, and if so, gives the UA packet to the application. Thus, Varuna delays the packet receipt by the higher layer. There are two types of Varuna delays—*steady delay* and *MOODy delay*. Steady delay is the delay when the source of the UA packet exists in the neighbor table. MOODy delay is the delay when the source does not exist in the neighbor table, causing the node to go to the MOODy state and verify its metadata with the source of the UA packet. Figure 7.6-d shows average steady delay for various neighbor table sizes. This delay is very small because the node just needs to scan its neighbor table in which it finds the source of the UA packet. This can be done in few microseconds as TelosB node has a 8MHz microcontroller. As the size of the neighbor table increases, the steady delay increases because the table look-up needs more time.

Figure 7.6-e shows the average MOODy delay for various values of the MOODy state parameters—ADV_RAND and τ. Recollect that when a node goes to the MOODy state, it waits for a random duration uniformly distributed between 0 and ADV_RAND before broadcasting the advertisement message. In the MOODy state, it advertises every τ time interval as long as it does not receive response from the node that caused it to be MOODy, or some threshold time interval T_{MOODy} expires. As expected, the increase in (ADV_RAND,τ) values increases the MOODy delay. Also, MOODy delay is higher for the sparse network than the dense network because poorer links in the sparse network causes more advertisement retransmissions in the

MOODy state, thereby increasing the MOODy delay. For d =5ft, ADV_RAND is the dominant factor in the MOODy delay. The MOODy delay is less than 12 seconds in our experiments. Also, initially many packets face the larger MOODy delay, but as neighbors are added to the neighbor table, the packets increasingly face the smaller steady delay. This is illustrated in Figure 7.6-f which shows that the ratio of the UA packets that face small steady delay to large MOODy delay increases linearly with the steady state time. Thus, overall Varuna introduces negligible delay while passing the received UA packet from MAC to the application.

7.4.2 Simulation Results

Use of sufficient memory for state maintenance is critically important in Varuna. As illustrated by testbed experiments, if sufficient memory is allocated for neighbor table, the steady state energy cost is fixed, independent of the steady state time period. In small and moderate size networks, the size of the neighbor table can be made sufficiently large. In order to find the appropriate size of the neighbor table to ensure fixed steady energy cost in large networks, we perform TOSSIM [127] simulations on a 20x20 grid network. Distance d between the grid points is taken as 5ft, 10ft, and 20ft. The values of various parameters used for the simulation experiments are same as those used for testbed experiments (Table 7.1). In our simulations, all UA packets are broadcast. In practice, not all UA traffic is of broadcast nature. If the amount of broadcast traffic is less, then as explained in Section 7.3.4, the number of MOODy transitions of the node, steady state energy cost, and the necessary neighbor table size are all reduced.

As Figure 7.7-a shows, the steady state energy cost increases linearly with time in Trickle, whereas it is fixed in Varuna for d =10ft and 20ft, for a neighbor table size of 50 slots. Trickle consumes about 5 and 147 times more energy than Varuna in one day and one month, respectively, for d =20ft. For a dense network with d=5ft, Figure 7.7-b shows that a neighbor table of 100 slots is required to achieve fixed steady state

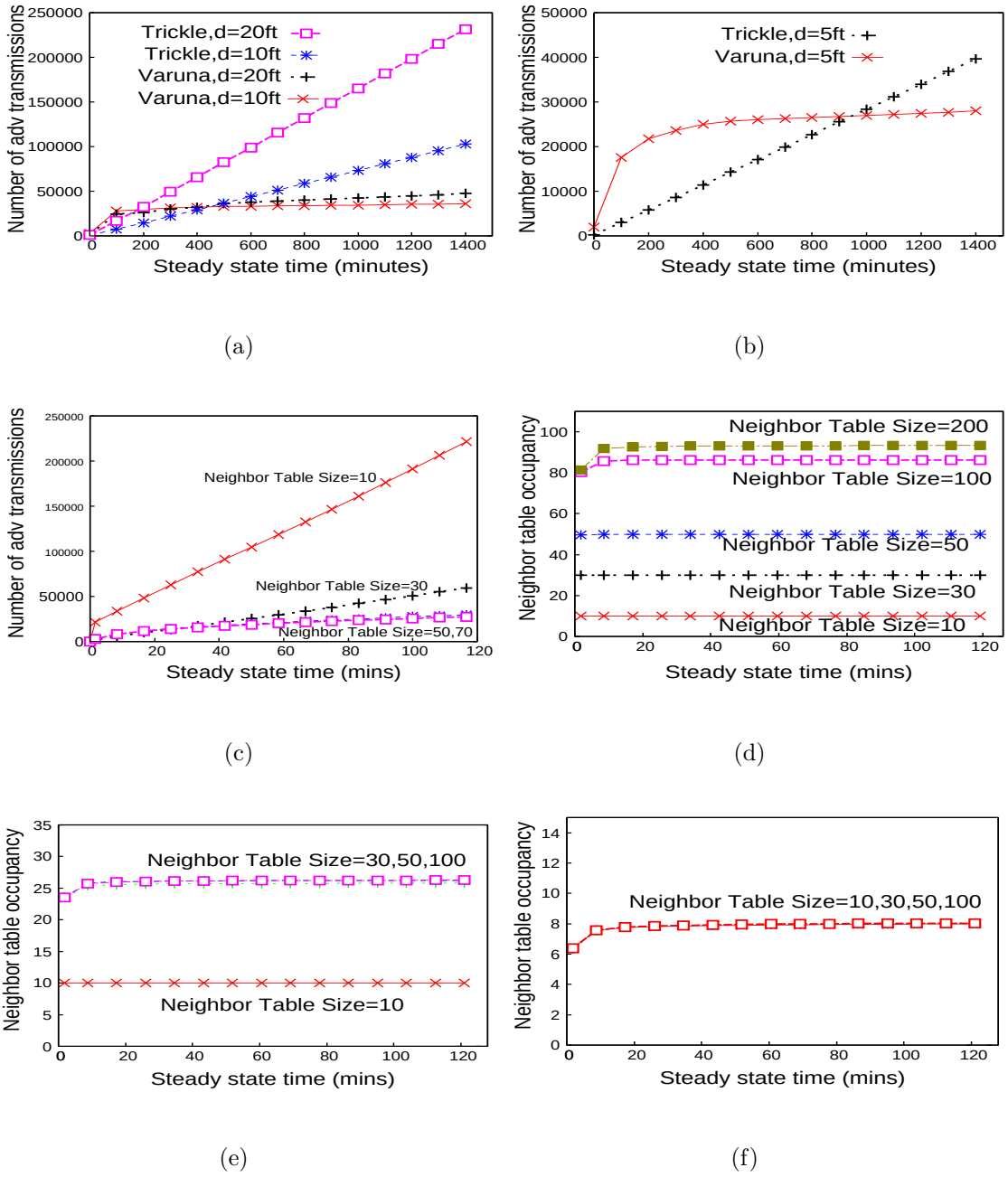


Fig. 7.7. Simulation results: Steady state energy cost as a function of time for (a) neighbor table size=50 and (b) neighbor table size=100; (c) Steady state energy cost for different neighbor table sizes for d=10ft; Neighbor table occupancy vs time for (d) d=5ft, (e) d=10ft, and (f) d=20ft.

energy cost in Varuna. For the dense network, Trickle's performance is better than for the sparse network because of its good advertisement suppression mechanism. But as the steady state time increases, Varuna outperforms Trickle. Figure 7.7-c shows the steady state energy cost for various values of neighbor table size and d =10 ft. Identical to the conclusion from the testbed experiments, if the neighbor table is kept sufficiently large, the steady state energy cost becomes independent of time in Varuna.

Figure 7.7-d, e, and f show the actual neighbor table size occupancy for d =5ft, 10ft, and 20ft. For sparse network with d =20ft, neighbor table with less than 10 slots is sufficient. For a denser network with d =10ft, less than 30 slots are sufficient. For a very dense network with d =5ft (and the large 400 node network), less than 100 slots (i.e. less than 600 bytes) are sufficient.

In the TOSSIM simulation model, each node has a transmission radius of 50ft, and the bit error rates are modeled using the empirical results from TinyOS experiments. To evaluate the size of the neighbor table necessary to ensure fixed steady state energy cost for different node densities, we introduce the notion of *density factor*. It is the ratio of actual network node density (nodes/ft^2) to *connectivity density* (nodes/ft^2). Connectivity density is the node density in a *minimally connected network* where every pair of neighboring nodes are at the farthest possible distance that allows direct one-hop communication between them. For example, in the grid network for our TOSSIM simulation, connectivity density is 4/2500 nodes/ft^2 since a square of 50ft side requires four nodes at four corners of the square to be minimally connected. Thus, density factor is also a measure of node redundancy in the network. For example, if density factor is k, then the network has k times more nodes than that minimally required for connectivity. For our experiments with d =20ft, 10ft, and 5ft, density factors are 6.25, 25, and 100, respectively. Note that because of failures and the fact that sensing range is generally smaller than the transmission range, a minimally connected network is generally not suitable for practical deployments. Nevertheless, a redundancy of factor 100 (for d =5ft) will likely be more than sufficient for most

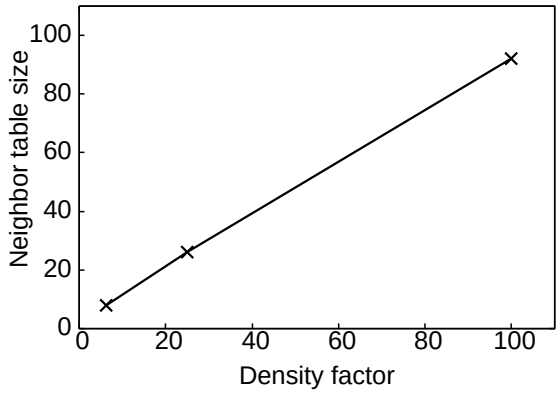


Fig. 7.8. Neighbor table size as a function of density factor.

deployments. For such a deployment, a neighbor table of 100 slots is sufficient. Figure 7.8 shows how the size of the neighbor table required for fixed steady state energy cost in Varuna increases with the density factor.

7.5 Related Work

Reliable data dissemination in wireless networks with unreliable communication characteristics has been the focus of several research efforts. For disseminating data-items to the nodes in wireless networks, [148] showed that indiscriminate flooding results in the highly resource-inefficient broadcast storm problem. Controlled flooding [149], gossiping [140] and probabilistic flooding [103] algorithms have been proposed for one-to-many communications in wireless ad-hoc networks to address this inefficiency.

Dissemination is more challenging in sensor networks where transient failures occur more frequently and unpredictably than in other types of wireless networks. Also energy consumption is a major concern as the sensor nodes have limited energy supply. Various protocols have been proposed for disseminating different types of data-items in sensor networks. Drip [150] disseminates various network parameters to the network nodes; Marionette [62] distributes network queries for debugging; [88, 92, 102] disseminate code binaries; Tenet [151] disseminates tasks. All these protocols use

Trickle [107] or some modified form of the Trickle algorithm. Trickle's design achieves fast and energy efficient distribution of the data-items in the dissemination phase under varying node densities. Dip [141] reduces the size of the metadata that has to be transmitted in an advertisement message if the information to be disseminated consists of several sub data-items. However the steady state energy expenditure of *all* of these protocols increases linearly with the steady state time, the most dominant phase in the lifetime of a network. To the best of our knowledge, Varuna is the first protocol to address the issue of steady state resource expenditure.

7.6 Conclusions

To maintain data-item consistency, existing systems cause nodes in the network to advertise their metadata periodically in the steady state when no dissemination is actually being done. In this chapter, we identify this as a severe problem—the steady state cost is overwhelmingly larger than the cost during the dissemination state. The steady state cost of all existing approaches increases linearly with the steady state time, which is the most dominant part of a network's lifetime. We presented the design and implementation of Varuna, whose steady state cost is independent of the steady state interval for most practical cases. Varuna achieves energy savings of several orders of magnitude compared to the existing standard algorithm called Trickle, as demonstrated through our testbed and simulation results.

The price that Varuna pays for achieving fixed steady state energy cost are memory requirement for storing neighbor table and guarantee of a relaxed invariant that information cannot flow from a higher version node to a lower version node. As is evident from our experiments, even for a very dense network with 100 times more nodes than that required for minimal connectivity, the memory requirement is reasonable for the currently available commercial sensor nodes. Also, in most sensor network deployments, the flow of information from up-to-date nodes to out-of-date nodes for some time (before it is eventually detected by Varuna when the information flow in

the opposite direction is attempted) is acceptable. The possibly erroneous information does not affect the up-to-date nodes, and cannot force wrong decisions at the base station, which can be assumed to be always up-to-date. Furthermore, in Trickle also, the information can flow in both directions (up-to-date to out of date and vice versa) for some time. Thus we believe that it is worthwhile to pay these relatively less important costs to ensure energy savings of several orders of magnitude, which proportionally increases the lifetime of the network.

8. CONCLUSIONS

The ability to reprogram the sensor nodes *in situ* using wireless medium is a very important requirement for the management of large scale sensor networks. The amount of information (often in kilobytes) that needs to be disseminated through the wireless channels for reprogramming purpose can be very large in the context of sensor networks where nodes typically use radio communications for exchanging small packets containing few bytes of sensed data. Wireless dissemination of software code updates can thus be very expensive in terms of energy consumption because radio transmissions are the most energy-intensive operations in typical sensor networks. Sensor nodes have limited energy supply and often need to be operated without changing batteries for long periods of time. Another important metric for wireless software update is the time required to reprogram the network. Since the performance of the sensor network may be degraded during reprogramming, it should be done quickly so that the nodes can resume their normal functions. This dissertation proposed some robust middleware services for sensor networks, that wirelessly reprogram the sensor network *quickly*, using less *energy* and *bandwidth*.

First, this thesis presented *Stream* that minimizes the amount of information that needs to be disseminated for reprogramming by partitioning the software running on the sensor nodes in two parts—reprogramming component and user application. By equipping the sensor nodes with the feature to load the reprogramming component on demand, Stream avoids the dissemination of the reprogramming component every time software update is done. This technique achieves a reduction of about 50% in both reprogramming time and energy compared to the state of the art scheme (Deluge) that bundles the reprogramming component and the user application into one image and disseminates the entire image. Stream also minimizes energy consumption

by putting nodes to sleep during the time period until the new code arrives at their neighborhood.

In spite of the current research trend towards multi hop reprogramming, in our next work *DStream* we use mathematical analysis, testbed experiments and simulations to show that under certain network conditions (low link qualities and small node degree), single hop reprogramming can be faster and more energy efficient than multi hop mode. DStream can switch between the two modes of reprogramming based on the current network conditions and thus save energy and increase network lifetime.

Reprogramming time and energy can be significantly reduced by wirelessly transferring the difference between the old and the new versions of the software (instead of the entire software) and letting the sensor nodes build the new software using the old software and the received difference. The difference between the currently executing code and the new code is typically much smaller than the entire code in practice. However, this dissertation shows that no matter how good a difference generator is, the difference will be disproportionately larger than the actual amount of change made in the software because of the shift of various software components. Even a small change in the software, for example, can cause many functions and global variables to be shifted in memory. As a result, all the instructions that refer to the shifted functions and variables change between the old and the new versions of the software, causing the difference to be large. This thesis proposed an incremental reprogramming system called *Zephyr* that, along with other features, mitigates the effect of function shifts, creating a small difference. We also presented *Hermes* that further reduces reprogramming time and energy by completely eliminating the effect of global variable shifts.

Finally this thesis presented *Varuna* that incurs fixed steady state energy cost independent of the steady state duration—time period during which no code update is being done in the network. Existing systems broadcast advertisement message periodically to make sure that all nodes are up-to-date all the time. This causes the energy expenditure in the steady state to increase linearly with the steady state

duration. Using the key observation that if the code running on the sensor node and neighborhood topology have not changed since the last advertisement, a node does not need to broadcast advertisement message, Varuna achieves energy savings of several orders of magnitude in the steady state comapared to existing systems. Varuna can also be used as a steady state maintenance protocol for information dissemination (not just code dissemination) in wireless ad-hoc networks (not just sensor networks).

LIST OF REFERENCES

LIST OF REFERENCES

[1] http://www.xbow.com/.

[2] http://www.ti.com/msp430/.

[3] http://www.sentilla.com/.

[4] http://www.eyes.eu.org.

[5] http://www.sunspotworld.com/.

[6] http://www.nanork.org/wiki/FireFly.

[7] http://www.tinynode.com/.

[8] http://www.sensinode.com/.

[9] http://www.redwirellc.com/.

[10] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "SOS: A dynamic operating system for sensor networks," *Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, pp. 163–176, 2005.

[11] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*, pp. 148–157, 2007.

[12] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[13] "The btnut operating system," in *http://www.vs.inf.ethz.ch*.

[14] http://www.tinyos.net/.

[15] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[16] J. Taneja, J. Jeong, and D. Culler, "Design, modeling, and capacity planning for micro-solar power sensor networks," in *Proceedings of the 7th international conference on Information processing in sensor networks (IPSN)*, pp. 407–418, IEEE Computer Society, 2008.

[17] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, *et al.*, "Operating a sensor network at 3500 m above sea level," in *Proc. of the 8th Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[18] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A macroscope in the redwoods," in *Proceedings of the 3rd international conference on Embedded networked sensor systems (Sensys)*, ACM, 2005.

[19] K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, "Deploying a wireless sensor network on an active volcano," *IEEE Internet Computing*, vol. 10, pp. 18–25, 2006.

[20] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, *et al.*, "PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes," in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 265–276, IEEE Computer Society, 2009.

[21] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler, "Lessons from a sensor network expedition," *Wireless Sensor Networks*, pp. 307–322.

[22] J. Koo, R. Panta, S. Bagchi, and L. Montestruque, "A tale of two synchronizing clocks," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 239–252, ACM, 2009.

[23] L. Mo, Y. He, Y. Liu, J. Zhao, S. Tang, X. Li, and G. Dai, "Canopy closure estimates with GreenOrbs: sustainable sensing in the forest," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 99–112, ACM, 2009.

[24] N. Ramanathan, T. Schoellhammer, E. Kohler, K. Whitehouse, T. Harmon, and D. Estrin, "Suelo: human-assisted sensing for exploratory soil monitoring studies," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 197–210, ACM, 2009.

[25] M. Keller, M. Yucel, and J. Beutel, "High-Resolution Imaging for Environmental Monitoring Applications," in *Proc. International Snow Science Workshop (ISSW Europe)*, vol. 10, Citeseer, 2009.

[26] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, "Health monitoring of civil infrastructures using wireless sensor networks," in *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*, ACM, 2007.

[27] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. 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 (Sensys)*, ACM, 2005.

[28] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 13–24, ACM, 2004.

[29] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA)*, pp. 88–97, ACM, 2002.

[30] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application driver for wireless communications technology," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2 supplement, p. 41, 2001.

[31] T. Liu, C. Sadler, P. Zhang, and M. Martonosi, "Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services (Mobisys)*.

[32] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 214–226, ACM, 2004.

[33] V. Singhvi, A. Krause, C. Guestrin, J. Garrett Jr, and H. Matthews, "Intelligent light control using sensor networks," in *Proceedings of the 3rd international conference on Embedded networked sensor systems (Sensys)*, ACM, 2005.

[34] X. Jiang, M. Van Ly, J. Taneja, P. Dutta, and D. Culler, "Experiences with a high-fidelity wireless building energy auditing network," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 113–126, ACM, 2009.

[35] S. Eisenman, E. Miluzzo, N. Lane, R. Peterson, G. Ahn, and A. Campbell, "BikeNet: A mobile sensing system for cyclist experience mapping," *ACM Transactions on Sensor Networks (TOSN)*, vol. 6, no. 1, pp. 1–39, 2009.

[36] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden, "CarTel: a distributed mobile sensor computing system," in *Proceedings of the 4th international conference on Embedded networked sensor systems (Sensys)*, ACM, 2006.

[37] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson, "VTrack: accurate, energy-aware road traffic delay estimation using mobile phones," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 85–98, ACM, 2009.

[38] C. Liang, J. Liu, L. Luo, A. Terzis, and F. Zhao, "RACNet: a high-fidelity data center sensing network," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, pp. 15–28, ACM, 2009.

[39] http://fire.me.berkeley.edu/.

[40] G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton, "Sensor network-based countersniper system," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 1–12, ACM, 2004.

[41] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, *et al.*, "Exscal: Elements of an extreme scale wireless sensor network," in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 102–108, 2005.

[42] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, and M. Gouda, "A line in the sand: A wireless sensor network for target detection, classification, and tracking," *Computer Networks*, vol. 46, no. 5, pp. 605–634, 2004.

[43] S. Patel, K. Lorincz, R. Hughes, N. Huggins, J. Growdon, D. Standaert, M. Akay, J. Dy, M. Welsh, and P. Bonato, "Monitoring motor fluctuations in patients with Parkinsons disease using wearable sensors," *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, no. 6, pp. 864–873, 2009.

[44] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton, "Codeblue: An ad hoc sensor network infrastructure for emergency medical care," in *International Workshop on Wearable and Implantable Body Sensor Networks*, vol. 5, Citeseer, 2004.

[45] B. Yang and S. Rhee, "Development of the ring sensor for healthcare automation," *Robotics and Autonomous Systems*, vol. 30, pp. 273–281, 2000.

[46] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter, "Luster: wireless sensor network for environmental research," in *Proceedings of the 5th international conference on Embedded networked sensor systems (Sensys)*, ACM, 2007.

[47] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin, "PermaSense: investigating permafrost with a WSN in the Swiss Alps," in *Proceedings of the 4th workshop on Embedded networked sensors*, p. 12, ACM, 2007.

[48] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 7, 2006.

[49] R. Murty, A. Gosain, M. Tierney, A. Brody, A. Fahad, J. Bers, and M. Welsh, "CitySense: A vision for an urban-scale wireless networking testbed," in *Proceedings of the 2008 IEEE International Conference on Technologies for Homeland Security, Waltham, MA*, Citeseer, 2008.

[50] R. Panta, S. Bagchi, I. Khalil, and L. Montestruque, "Single versus multi-hop wireless reprogramming in sensor networks," in *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities (Tridentcom)*, pp. 1—7, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[51] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems (Sensys)*.

[52] M. Wachs, J. Choi, J. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis, "Visibility: A new metric for protocol design," in *Proceedings of the 5th international conference on Embedded networked sensor systems (Sensys)*.

[53] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *20th International Parallel and Distributed Processing Symposium (IPDPS)*.

[54] A. Woo and T. Tong, "Tinyos mintroute collection protocol," in *tinyos-1.x/lib/MintRoute*.

[55] T. Van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems (Sensys)*.

[56] R. Beckwith, D. Teibel, P. Bowen, and I. Res, "Unwired wine: Sensor networks in vineyards," in *Proceedings of IEEE Sensors*, pp. 561–564, 2004.

[57] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden, "Task: Sensor network in a box," in *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[58] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. Stankovic, *et al.*, "VigilNet: An integrated sensor network system for energy-efficient surveillance," *ACM Transactions on Sensor Networks (TOSN)*, vol. 2, no. 1.

[59] T. Schmid, H. Dubois-Ferriere, and M. Vetterli, "Sensorscope: Experiences with a wireless building monitoring sensor network," in *Workshop on Real-World Wireless Sensor Networks (REALWSN05)*, 2005.

[60] V. Turau, C. Renner, M. Venzke, S. Waschik, C. Weyer, and M. Witt, "The heathland experiment: Results and experiences," in *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN)*, Citeseer, 2005.

[61] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh, "Monitoring volcanic eruptions with a wireless sensor network," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, pp. 108–120, 2005.

[62] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: using rpc for interactive development and debugging of wireless embedded networks," in *Proceedings of the 5th international conference on Information processing in sensor networks (IPSN)*.

[63] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proceedings of the 3rd international conference on Embedded networked sensor systems (Sensys)*.

[64] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 201–213, ACM, 2004.

[65] P. Li and J. Regehr, "T-Check: Bug Finding for Sensor Networks,"

[66] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula, "Dependent types for low-level programming," *Programming Languages and Systems*, pp. 520–535, 2007.

[67] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for TinyOS," in *Proceedings of the 5th international conference on Embedded networked sensor systems (Sensys)*.

[68] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems (Sensys)*, pp. 85–98, ACM New York, NY, USA, 2008.

[69] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems (Sensys)*, pp. 99–112, ACM, 2008.

[70] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from TinyOS programs using symbolic execution," in *Proceedings of the 7th international conference on Information processing in sensor networks (IPSN)*, pp. 271–282, IEEE Computer Society, 2008.

[71] V. Krunic, E. Trumpler, and R. Han, "NodeMD: Diagnosing node-level faults in remote wireless sensor systems," in *Proceedings of the international conference on Mobile systems, applications, and services (MobiSys)*, pp. 43–56, ACM, 2007.

[72] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: software-based memory protection for sensor nodes," in *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*.

[73] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with envirolog," tech. rep., Citeseer, 2006.

[74] N. Nguyen and M. Soffa, "Program representations for testing wireless sensor network applications," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*.

[75] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment," in *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.

[76] I. S. 802.15.4, "Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)," 2006.

[77] I. S. 802.11, "IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," 2007.

[78] I. S. 802.15.4, "Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks," 2005.

[79] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *Proceedings of the 6th ACM conference on Embedded network sensor systems (Sensys)*, pp. 43–56, ACM, 2008.

[80] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving sensor network software faults," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pp. 235–246, ACM, 2009.

[81] http://www.heliosware.com/.

[82] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: update-conscious compilation for energy efficiency in wireless sensor networks," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*.

[83] C. Fok, G. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 653–662, 2005.

[84] http://ceti.cse.ohio-state.edu/kansei/.

[85] http://www.jamesreserve.edu/.

[86] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 188–200, ACM, 2004.

[87] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pp. 60–67, 2003.

[88] R. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *IEEE Conference on Computer Communications (Infocom)*, pp. 928–936, 2007.

[89] R. Panta, S. Bagchi, and I. Khalil, "Efficient wireless reprogramming through reduced bandwidth usage and opportunistic sleeping," *Ad Hoc Networks*, vol. 7, no. 1, pp. 42–62, 2009.

[90] R. Panta, S. Bagchi, and S. Midkiff, "Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation," in *USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[91] R. Panta and S. Bagchi, "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks," in *IEEE Conference on Computer Communications (Infocom)*, 2009.

[92] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *Proceedings of the international conference on Embedded networked sensor systems (Sensys)*, pp. 81–94, 2004.

[93] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," 1999.

[94] J. Lilius and I. Paltor, "Deeply embedded python, a virtual machine for embedded systems," in *http://www.tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy*.

[95] A. Boulis, C. Han, and M. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, pp. 187–200, ACM, 2003.

[96] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *ACM SIGOPS Operating Systems Review*, pp. 85–95, 2002.

[97] J. Koshy and R. Pandey, "VMSTAR: synthesizing scalable runtime environments for sensor networks," *Proceedings of the international conference on Embedded networked sensor systems (Sensys)*, pp. 243–254, 2005.

[98] P. Levis, D. Gay, and D. Culler, "Active sensor networks," *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[99] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. Sirer, "Design and implementation of a single system image operating system for ad hoc networks," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys)*, pp. 06–08, 2005.

[100] C. Inc, "Mote In-Network Programming User Reference," 2003.

[101] S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," vol. 25.

[102] M. Krasniewski, R. Panta, S. Bagchi, C. Yang, and W. Chappell, "Energy-efficient on-demand reprogramming of large-scale sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, 2008.

[103] J. Luo, P. Eugster, and J. Hubaux, "Route driven gossip: probabilistic reliable multicast in ad hoc networks," *IEEE Conference on Computer Communications (Infocom)*, pp. 2229–2239, 2003.

[104] J. Kulik, W. Heinzelman, and H. Balakrishnan, "Negotiation-based protocols for disseminating information in wireless sensor networks," *Wireless Networks*, vol. 8, no. 2, pp. 169–185, 2002.

[105] G. Khanna, S. Bagchi, and Y. Wu, "Fault tolerant energy aware data dissemination protocol in sensor networks," in *International Conference on Dependable Systems and Networks (DSN)*, pp. 795–804, 2004.

[106] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," *University of California, LA, Tech. Rep. CENS-TR-30*, 2003.

[107] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 246, pp. 15–28.

[108] R. Simon, L. Huang, E. Farrugia, and S. Setia, "Using multiple communication channels for efficient data dissemination in wireless sensor networks," in *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference (MASS)*, 2005.

[109] P. Dutta, J. Hui, D. Chu, and D. Culler, "Securing the deluge network programming system," in *Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 326–333, 2006.

[110] P. Lanigan, R. Gandhi, and P. Narasimhan, "Sluice: Secure dissemination of code updates in sensor networks," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[111] H. Tan, S. Jha, D. Ostry, J. Zic, and V. Sivaraman, "Secure multi-hop network programming with multiple one-way key chains," in *Proceedings of the first ACM conference on Wireless network security*, pp. 183–193, ACM, 2008.

[112] S. Hyun, P. Ning, A. Liu, and W. Du, "Seluge: Secure and dos-resistant code dissemination in wireless sensor networks," in *Proceedings of the international conference on Information processing in sensor network (IPSN)*, pp. 445–456, 2008.

[113] H. Tan, D. Ostry, J. Zic, and S. Jha, "A confidential and DoS-resistant multi-hop code dissemination protocol for wireless sensor networks," in *Proceedings of the second ACM conference on Wireless network security*, pp. 245–252, ACM New York, NY, USA, 2009.

[114] O. Ugus, D. Westhoff, and J. Bohli, "A ROM-friendly secure code update mechanism for WSNs using a stateful-verifier τ-time signature scheme," in *Proceedings of the second ACM conference on Wireless network security*, pp. 29–40, ACM, 2009.

[115] C. de la Parra and J. Garcia-Macias, "A protocol for secure and energy-aware reprogramming in WSN," in *Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly*, pp. 292–297, ACM, 2009.

[116] W. Itani, A. Kayssi, and A. Chehab, "PETRA: a secure and energy-efficient software update protocol for severely-constrained network devices," in *Proceedings of the 5th ACM symposium on QoS and security for wireless and mobile networks*, pp. 37–43, ACM, 2009.

[117] W. Li, Y. Zhang, and B. Childers, "MCP: An Energy-Efficient Code Distribution Protocol for Multi-Application WSNs," in *Proceedings of 5th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, Springer-Verlag New York Inc, 2009.

[118] W. Li, Y. Du, Y. Zhang, B. Childers, P. Zhou, and J. Yang, "Adaptive Buffer Management for Efficient Code Dissemination in Multi-Application Wireless Sensor Networks," in *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008. EUC'08*, vol. 1, 2008.

[119] D. De Couto, D. Aguayo, B. Chambers, and R. Morris, "Performance of multi-hop wireless networks: Shortest path is not enough," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1.

[120] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming wireless sensor networks: challenges and approaches," *IEEE Network*, vol. 20, no. 3, pp. 48–55, 2006.

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

[122] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, pp. 354–365, 2005.

[123] P. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," *Wireless Sensor Networks*, pp. 212–227.

[124] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. Baras, "Atemu: A fine-grained sensor network simulator," in *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pp. 145–152, 2004.

[125] W. Li, Y. Zhang, J. Yang, and J. Zheng, "Towards update-conscious compilation for energy-efficient code dissemination in WSNs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 4, pp. 1–33, 2009.

[126] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*.

[127] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire tinyOS applications," *Proceedings of the international conference on Embedded networked sensor systems (Sensys)*, pp. 126–137, 2003.

[128] S. Bandyopadhyay and E. Coyle, "An energy efficient hierarchical clustering algorithm for wireless sensor networks," in *IEEE Conference on Computer Communications (Infocom)*, vol. 3, pp. 1713–1723, 2003.

[129] J. Kim and J. Lee, "Performance analysis of Mac protocols for wireless LAN in Rayleigh and shadow fast fading," in *IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 1, pp. 404–408, 1997.

[130] http://estadium.purdue.edu/.

[131] H. Pucha, D. Andersen, and M. Kaminsky, "Exploiting similarity for multi-source downloads using file handprints," *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[132] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, pp. 174–187, 2001.

[133] http://jarsync.sourceforge.net/.

[134] A. Kamra, V. Misra, J. Feldman, and D. Rubenstein, "Growth codes: maximizing sensor network data persistence," *Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pp. 255–266, 2006.

[135] M. Buettner, G. Yee, E. Anderson, and R. Han, "X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks," in *ACM 4th international conference on Embedded networked sensor systems (Sensys)*, 2006.

[136] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *ACM 2nd international conference on Embedded networked sensor systems (Sensys)*, pp. 95–107, 2004.

[137] A. El-Hoiydi and J. Decotignie, "Low power downlink mac protocols for infrastructure wireless sensor networks," *Mobile Networks and Applications*, pp. 675–690, 2005.

[138] cc2420, "http://focus.ti.com/lit/ds/symlink/cc2420.pdf."

[139] http://www.atmel.com/.

[140] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," *ACM Transactions on Networking (TON)*, pp. 767–780, 2004.

[141] K. Lin and P. Levis, "Data discovery and dissemination with dip," *Proc. of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 433–444.

[142] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[143] "Wireless systems for industrial automation, process control, and related applications," *ISA 100.11a, Draft standard*, 2009.

[144] G. Ahn, S. Hong, E. Miluzzo, A. Campbell, and F. Cuomo, "Funneling-mac: a localized, sink-oriented mac for boosting fidelity in sensor networks," in *Proceedings of the international conference on Embedded networked sensor systems (Sensys)*, 2006.

[145] C. Perkins, E. Royer, and S. Das, "Ad hoc on-demand distance vector (AODV)," in *IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 1999.

[146] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," *Internet proposed standard RFC 4944*, 2007.

[147] "http://www.zigbee.org," *ZigBee Allinance*.

[148] S. Ni, Y. Tseng, Y. Chen, and J. Sheu, "The broadcast storm problem in a mobile ad hoc network," in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking (Mobicom)*, pp. 151–162, ACM, 1999.

[149] S. Floyd, V. Jacobson, S. McCanne, C. Liu, and L. Zhang, "A Reliable Multicast Framework for light-weight Sessions and Application Level Framing," *Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM*, pp. 342–356, 1995.

[150] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, pp. 121–132, 2005.

[151] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," *Proceedings of the international conference on Embedded networked sensor systems (Sensys)*, pp. 153–166, 2006.

VITA

VITA

Rajesh Krishna Panta received B.E. in Electronics Engineering from Tribhuwan University, Nepal. He obtained Masters degree in Information Engineering from Niigata University, Japan, under Japanese government scholarship. He started Ph.D. program in Purdue in Fall, 2005.

Rajesh's research interests are in broad areas of embedded, networked, and distributed systems, with an emphasis on wireless ad-hoc and sensor networks. He worked in Robert Bosch Research and Technology Center during summer 2009.