This is to certify that the
dissertation entitled

ENERGY EFFICIENT REPROGRAMMING FOR SENSOR NETWORKS

presented by

Limin Wang

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science and Engineering

SSKulkarni
Major Professor's Signature

June 20, 07
Date

ENERGY EFFICIENT
REPROGRAMMING FOR SENSOR NETWORKS

By

Limin Want

A DISSERTATION

Submitted to
Michigan State University
In partial fulfillment of the requirements
For the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2007

ABSTRACT

ENERGY EFFICIENT REPROGRAMMING FOR SENSOR NETWORKS

By

Limin Wang

In this dissertation, we focus on the problem of reprogramming for sensor networks, which is an important and challenging problem as it is often necessary to reprogram the sensors in place. Reprogramming can be done in two ways. For one, one (or a few) sensor that has the entire program is dropped to the field. The sensor then propagates the program to the remaining sensors in the network. We propose MNP, a multihop reprogramming protocol, for this scenario. Another way of reprogramming is to assume that each sensor (or a subset of the sensors) in the network has a part of the program initially. These sensors gossip among each other to get the remaining parts of the program. We propose *Gappa*, a gossip-based multi-channel reprogramming protocol, for this gossip among sensors.

Reprogramming of sensor networks needs to meet the following requirements. First, reprogramming requires 100% reliability. Second, energy efficiency is important in the context of sensor networks. For reprogramming, idle listening and message collision are the two major sources of energy waste, and hence, must be reduced. Third, if sensors are deployed in an untrusted environment, they must be able to verify that the code is truely from a trusted source, i.e., authentication for reprogramming is needed. We address these requirements for the two reprogramming scenarios mentioned above.

First, MNP and *Gappa* use an automatic repeat request (ARQ) scheme to provide reliability. In this scheme, a receiver detects its own losses, and informs the sender of the missing packets. The sender then retransmits the packets requested by the

receivers. Another way of providing reliability is to use forward error correction (FEC). We also propose a hybrid scheme which combines ARQ and FEC techniques to provide reliability in reprogramming protocols. We apply it to MNP and find that using FEC can improve reprogramming performance.

Second, MNP uses a sender selection algorithm to reduce message collision. The sender selection algorithm tries to guarantee that there is only one sender in a neighborhood at a time. *Gappa* extends the sender selection algorithm from MNP to multiple radio channels. If a sensor loses sender selection on one channel, it completes to transmit code on a different channel that is available. For *Gappa*, we propose two policies for assigning radio channels, fixed channel allocation and variable channel allocation. We note that the variable channel allocation scheme allows better utilization of the available channels, and hence, performs better than the simple fixed channel allocation scheme. In MNP and *Gappa*, the sensors that lose in the sender selection algorithm and are not interested in receiving code from its neighbors are put to sleep. In this way, MNP and *Gappa* effectively reduce the active radio time of sensors and save energy.

Finally, to provide authentication to reprogramming protocols such as MNP, we propose a symmetric key based authentication protocol. This protocol is based on the secret instantiation algorithm that requires only $O(\log n)$ keys to be maintained at each sensor. We show that the use of symmetric keys can significantly reduce the cost of authentication compared to public key based schemes, especially in the cases where moderate amount of data needs to be disseminated. We also propose additional schemes to reduce the cost of secure data dissemination by adding redundancy to the transmitted data.

Dedicated to my parents and my sister for their love and support.

ACKNOWLEDGMENTS

I owe my gratitude to many people that have assisted me along the way of pursuing a PhD degree. They have made my graduate experience the one that I will cherish forever.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Sandeep S. Kulkarni, for his guidance, encouragement, and support for the past 5 years. Sandeep has been a wonderful advisor. He is brilliant, resourceful, thoughtful, supportive, patient, and cheerful. He is the one who inspired me to explore interesting research problems, and guided me on the steps to achieve my goals. I appreciate the countless hours he has spent with me discussing my research, commenting on my writings, and critiquing my talks. I am grateful to him not only for his contribution to this dissertation but also for the outstanding mentorship that he has provided me.

I would like to thank Dr. Betty H.C. Cheng, Dr. Li Xiao, and Dr. Rajesh Kulkarni for serving in my dissertation committee. They gave me valuable comments on my dissertation proposal and provided me important suggestions on extending the proposed research.

I would like to thank all the faculty and staff in the Department of Computer Science and Engineering at Michigan State University. I am thankful to Dr. Philip K. McKinley and Dr. Alex X. Liu for taking a special interest in my research and giving me valuable comments. I would like to express my gratitude to Dr. George C. Stockman for his help in TA assignments, his care and encouragement. I am also thankful to Dr. Eric Torng for his assistance in my application for Dissertation Completion Fellowship. Also, thank Ms. Linda Moore for her patience and cheerfulness when performing administrative tasks.

As a member of the Software Engineering and Network Systems (SENS) Laboratory, I want to express my gratitude to all the SENS faculty and students for providing such a stimulating environment to work in. I benefit from the seminars and discussions in this group. I am thankful to Dr. Mahesh Arumugam, Borzoo Bonakdarpour, Dr. Bru Bezawada, Dr. Sascha Konrad, Dr. Ali Ebnenasir, Karli Lopez, Dr. Ji Zhang, Karun Biyani, Fuad Suleiman Abujarad, and Chad R. Meiners for their help in proof reading my papers and providing constructive comments on improving my presentations. I appreciate the friendship and generous help from Min Deng, Chiping Tang, Eduardo Diaz-Sanchez, Yun Zhou, and Yi Huang.

Thank my advisor for providing me the opportunity to participate in the ExScal project (funded by Defense Advanced Research Projects Agency (DARPA)), in which I get to work with the professors and students from many universities. I would like to express my sincere thanks to Dr. Anish Arora (at the Ohio State University), Dr. Rajiv Ramnath (at the Ohio State University), Dr. Mohamed Gouda (at the University of Texas at Austin), Dr. Mikhail Nesterenko (at the Kent State University), and Dr. Ted Herman (at the University of Iowa) for their generous help and important suggestions on my work. I also want to thank Young-ri Choi, Dr. Hongwei Zhang, Dr. Sandip Bapat, Vinod Kulathumani, Dr. Vinayak Naik, Mukundan Sridharan, Dr. Santosh Kumar, and Chen Zhang for their help and friendship. We have spent the memorable two weeks together preparing for DARPA field demonstration in Avon Park, FL, in December 2004.

I want to thank all my friends at Michigan State University. Special thanks to Jim Maher and John Oleszkiewicz. I am so fortunate to have them as my friends. They made my life full of fun and laughter. I also want to thank my friendship family, Bonnie and John Bankson, for their care and support.

Last but not the least, I would like to thank my parents and my sister for their constant love, care and support. None of my achievements would have been possible without the love and faith from my family.

Table of Contents

List of Tables

LIST OF FIGURES

Chapter 1

Introduction

The recent advances in Micro-Electro-Mechanical Systems (MEMS) and low power radio technologies have enabled the deployment of wireless sensor networks for a wide variety of applications. These applications include intrusion detection and tracking (e.g., A Line in the Sand [1], ExScal [2]), habitat monitoring (e.g., monitoring seabird nesting behavior [44]), indoor surveillance (e.g., provide security in an art gallery), hazard detection (e.g., forest fire detection), traffic analysis (e.g., monitor vehicle traffic on a highway or a congested part of a city).

In these applications, the network is composed of a large number of densely deployed, small, low-power devices, called sensors. The sensors are capable of (a) sampling the physical environment, (b) performing local processing and computing on the sampled data, which include acoustic, temperature, light, magnetic, and other chemical and physical properties, (c) communicating the sampled/processed data to other sensors or the base station through radio. Each sensor, due to small form-factor, has limited processing capability and power supply, and can communicate only within short distances. However, when coordinated with a large number of other sensors, they have the ability to monitor the physical environment in great detail.

1.1 Background: Sensor Networks Characteristics and Challenges

Sensor networks offer tremendous technology possibilities, and, at the same time, pose a number of new challenges. The characteristics and challenges of sensor networks are discussed below.

Sensors have **limited computation resources (e.g., processor, memory), limited communication bandwidth, and limited power supply**. In Table 1.1, we list the hardware constraints of some existing sensor platforms. Specifically, the sensors listed in Table 1.1 have only 4-10 KB RAM, which is shared among all the operating components including the kernel and some application specific components such as routing, synchronization, reprogramming, localization, sensing. The communication bandwidth is limited up to 250 kbps. Compared to other resource constraints, energy is a even more scarce resource for sensors. Since sensors are normally powered by batteries that are difficult to replace after they are deployed, it is critical to design **power management algorithm** to utilize minimal energy. We will discuss energy conservation for sensor networks in more details in Section 1.2.

In most situations, sensors, once deployed, are expected to operate unattended for a long time. This leads to the problem of **maintenance and reconfiguration**

Table 1.1: Hardware configuration of some existing sensor platforms

Platform	**Processor**	**Program memory**	**RAM**	**Non-volatile storage**	**Radio bandwidth**
Mica2 [7] and XSM [10]	8-bit, 7 MHz ATmega128L	128 KB	4 KB	512 KB	19.2 kbps (Manchester encoded)
MicaZ [8]	8-bit, 7 MHz ATmega128L	128 KB	4 KB	512 KB	250 kbps
TolesB [11]	16-bit, 8 MHz TI MSP430	48 KB	10 KB	1024 KB	250 kbps

of the sensors. As the environment evolves over time, predicting the whole set of actions that a sensor might need to perform is impossible in most applications. The requirements are also likely to change. For example, with growing understanding of the environment or with new technological advances, some assumptions are found to be incorrect, and hence, the specification has to be modified accordingly. Hence, reprogramming a sensor network, i.e., sending software updates to sensors to fix bugs and/or improve features, is necessary. In Section 1.3, we will discuss the challenges in network reprogramming in detail.

Due to the fact that sensor networks are often applied in a lot of domains that handles sensitive information, there is a need to develop security approaches to protect the sensitive data from being disclosed to unauthorized third parties. Security objectives for sensor networks are the same as those for traditional networks, which include confidentiality (ensures that sensitive data is not revealed to unauthorized third party), authentication (verifies that the data received is really sent by a trusted sender), integrity (verifies that the data received is not altered when exchanged over insecure networks), freshness (ensures that messages are fresh, i.e., not duplicated or replayed by an adversary), availability (resilience to denial of service attacks). However, it is not efficient, if not impractical, to apply the traditional security solutions to the resource-starved sensor nodes. For example, most authentication approaches reply on asymmetric digital signatures, which typically require long signatures with high communication overhead of 50-1000 bytes per packet and very complex processing for generating and verifying the signatures. Key establishment and management is also difficult due to the energy, communication bandwidth, and storage constraints. It is necessary to design **light-weight, efficient security solutions** to address the requirements in sensor networks.

Scalability is an important issue in sensor networks. The power of wireless sensor networks lies in the ability to deploy a *large* number of small, low-cost sensors to achieve complex tasks. The size of sensor networks has increased substantially in the past few years. The largest sensor network that has been deployed so far is *ExScal* [2,4], in which about 1200 sensors are deployed over 1.3km by 300m open area. Such sensor networks must be scalable. Unlike traditional wireless system such as cell phone systems, where the quality of service degrades when there are too many wireless devices (phones) in a small area, adding more sensor nodes to a network should only improve the power of network, either by increasing the region that the network covers, or by enabling stronger interconnection of the wireless sensor network. Any form of central control point that would limit the number of working sensor nodes in a given area is undesirable in a wireless sensor network.

One reason that sensor networks gain tremendous popularity is their **low deployment cost** and their **ability to self-configure**. Unlike traditional networks, which require expensive wiring or building high-power base stations, sensor networks do not depend on pre-existing infrastructure, and can be deployed with minimal cost. A typical deployment scenario is to drop the sensors from an aircraft when it flies over the field. This is especially useful in some applications such as military surveillance or disaster recovery, where the target region is inaccessible to humans and/or a network for collecting critical data must be deployed in real-time.

After deployment, sensors must be able to self-configure and adapt to the environment. Instead of communicating directly to the base stations, each sensor communicates only to its local neighbors. Data flow to/from the base stations are relayed on thousands of tiny sensors in a multi-hop fashion.

Although the individual sensors are **prone to failures** (due to lack of power, physical damage), sensor networks can achieve high robustness by automatically replacing the failed sensors by their neighbors. Moreover, sensors communicate through **wireless radio**, which is **low-bandwidth and unreliable**. In most cases, loss of data is *not* considered as a serious problem as data from sensors that are densely deployed in the same region are highly redundant. However, for some crucial data, e.g., software updates, aggregated reports, **reliable communication** is important.

In Table 1.2, we summarize the characteristics and corresponding challenges in sensor networks.

1.2 Energy Conservation

Among all the resource constraints of a sensor node, the biggest one being power supply. The sensor platforms that are listed in Table 1.1 are all powered by 2 AA batteries. Due to the large scale of sensor networks and the embedded nature of sensor nodes, once the network is deployed, it is very difficult to change batteries for the sensors. Although recent work has shown that energy harvesting techniques, i.e., collecting the ambient energy from the environment, can significantly benefit some applications, they are limited by the efficiency of the transducers and the availability of the raw energy (e.g., solar, mechanical energy). Therefore, Energy conservation operations are crucial for extending network lifetime.

According to our experience, without any energy conservation approaches, a Mica2 mote transmitting a message every second can operate continuously for only

Table 1.2: Characteristics and Challenges in Sensor Networks

Characteristics	**Challenges**
Sensors have limited CPU, memory and radio bandwidth	Sensor network algorithms must have low resource requirements
Sensors have limited power supply	Power management
Sensors are left unattended for a long time	Maintenance and Reconfiguration
Sensors are often used to handle sensitive data	Security
Large number of sensors	Scalability
Low deployment cost	Ability to self-configure
Sensor are prone to failure	Robustness against node failure
Wireless media is unreliable	Reliable communication

about a week. However, if we reduce the frequency that sensors report data to once every 3 minutes and apply appropriate power management techniques [52, 73], the estimated lifetime is 1-3 years (328-945 days) [53].

Previous work [54] shows that radio communication is an energy expensive operation. Pottie and Kaiser [54] compared the cost of communication and computation and concluded that the energy cost of transmitting 1KB a distance of 100 meters is approximately the same as executing 3 million instructions. Hence, the key to reduce energy consumption is to minimize radio communication. A couple of techniques are introduced to reduce the amount of communication. By in-network processing and aggregation, sensors send data to the aggregation nodes, which perform local processing, and forward the aggregated/filtered data to the next hop. A similar technique is clustering, in which a cluster head collects data from the sensors in its cluster and forwards the processed data to a neighboring cluster until it reaches the base station.

Moreover, even reducing the amount of radio communication is not sufficient. In Table 1.3 (from [44]), we list the costs of various operations on a Mica mote with a pair of AA batteries. As we can see, the energy cost of leaving a sensor node in idle listening state for 1 second is the same as the energy cost of transmitting 62 messages. In fact, without power conservation approaches, more than 90% of the energy is wasted on idle listening where sensors do not produce any useful output [73]. Therefore, it is necessary to put sensors to sleep mode when they are idle and wake them up when required.

Table 1.3: Charge required by various Mica operations

Operation	nAh
Transmitting a packet	20.000
Receiving a packet	8.000
Idle listening for 1 millisecond	1.250
EEPROM Read 16 Bytes	1.111
EEPROM Write 16 Bytes	83.333

Energy conservation can be performed on different network layers. For example, S-MAC [73] and B-MAC [52] are two low power MAC layer protocols for sensor networks, and there are numerous energy aware routing protocols (e.g., [21, 74]). An energy conservation approach is most effective when it is incorporated in specific applications, as relevant application layer information can help sensors to decide when it should go to sleep and when to wake up. For example, several node scheduling approaches [19, 62, 69, 71, 72], including our recent work [3, 67], have been proposed for surveillance sensor networks to minimize energy consumption while maintaining sensing coverage at the desired level. These approaches try to maintain a subset of sensor nodes in working mode and put the remaining nodes to sleep, in order to achieve a long network lifetime. In this dissertation, we focus on energy conservation approaches during reprogramming.

1.3 Reprogramming for Sensor Networks

The success of sensor network technology hinges on its ease of maintenance. Although some changes can be done by varying configuration parameters, or downloading scripts [35], more significant changes require sending the new binary image of the program to all the sensors in the network. Due to the large scale of the sensor networks and the deployment in environment with high access cost, even collecting the sensors from the field for reprogramming is a daunting task. Therefore, reprogramming needs to be done in place, without physical contact with the sensors.

Reprogramming in sensor networks poses several new challenges and requirements. First, reprogramming requires 100 percent delivery. This is very different from traditional sensor network applications, in which, occasional loss of data is tolerable and often expected.

Second, high communication bandwidth is needed in network reprogramming.

For the vast majority of sensor network applications, the generated sensing data from an individual sensor node is small, usually of the order of bytes, and thus easily fits the low wireless radio bandwidth. However, network reprogramming requires delivery of the entire program image, of the order of kilobytes, over low-bandwidth wireless radio, and hence consumes significant bandwidth.

Third, the problem of concurrent senders needs to be addressed. In network reprogramming, code image is propagated from one sensor node to another. Every node that has the new code image is a potential sender. Thus, it is likely that several senders are transmitting at the same time. This causes a lot of message collisions and congestion of wireless channel, and possibly results in failure of reprogramming.

Fourth, energy efficiency is important. Because sensor nodes have limited power supply, the amount of energy consumed in network reprogramming may directly affect network lifetime. Some of the possible sources of energy inefficiency include message collision, overhearing, control message overhead, and idle listening. Among these, idle listening is the major source of energy waste, as we discussed in Section 1.1. Reducing the number of messages sent and received is also important.

Securing the process of network reprogramming is important in many scenarios. Sensors must verify authenticity, check integrity, ensuring freshness of the received program binary, before they boot to the new program. In some security intensive applications such as military surveillance, the program itself can be sensitive data, and must not be revealed to unauthorized third parties. In this case, confidentiality is also required. However, providing security for reprogramming is challenging due to the following reasons. First, since wireless radio is a broadcast media, an adversary can easily inject a packet which passes CRC check at the link layer. Such an attack must be detected immediately, otherwise the entire program that has been downloaded must be discarded. Second, due to the extreme resource constraints, the security solutions must use minimal computation, storage, and energy resources, as we mentioned in Section 1.1. Third, as the energy cost of receiving a packet, storing it to EEPROM and verify the program is high, an adversary can launch a Denial of Service attack that aims to drain sensors' power.

1.4 Requirements and System Model of Reprogramming

In this dissertation, we focus on the task of reprogramming for wireless sensor networks. Reprogramming for sensor networks needs to deal with all the challenges identified in Sections 1.1-1.3, i.e., it needs to be done in a reliable, energy efficient way, and it must scale to large networks. In the case that sensors are deployed in untrusted environments, authentication must be provided. Specifically, we identify the basic requirements that a reprogramming protocol should meet as follows:

1. *Reliability.* This includes both *accuracy* and *coverage* requirements. By *accuracy*, we mean that the *exact* program image is received by sensor nodes; and by *coverage*, we mean that eventually *every* sensor node in the network is reprogrammed with the new code.

2. *Autonomy.* Code should be propagated automatically, without human intervention.

3. *Authentication.* Sensors must be able to verify the authenticity and integrity of the received program.

4. *Energy efficiency.* The energy used in code dissemination should be low so as to affect the network lifetime minimally.

5. *Speed.* New program code should be propagated and installed quickly.

Among these requirements, reliability and autonomy are the basic and essential requirements for the correctness of a code dissemination protocol. Authentication is required if the sensors are deployed in an untrusted environment. Energy efficiency and speed are the two important metrics we use to evaluate the performance of a reprogramming protocol.

In all the protocols we propose in this dissertation, we make no assumptions about the underlying network topology. We only consider networks with stationary nodes. Sensor nodes do not need to have any location information or maintain neighbor status. Each sensor node makes local decisions independently.

We divide the program image into *segments*, each of which contains a fixed number of packets (with the exception of the last segment). Each segment is assigned an ID that is strictly increasing. A sensor can advertise/transmit a segment only when it has received the entire segment.

Moreover, we assume that reprogramming service has full control of a node's resources (including radio and hardware) during reprogramming (from the point that a node receives a reprogramming signal (advertisement, request, data packet, etc.). When a node reboots to a new program, or rolls back to the old program (because of interruption or error during reprogramming), the application has control of the resources.

1.5 Contributions of The Dissertation

In this dissertation, we defend the following thesis:

The sender selection algorithm that attempts to select one sender in a neighborhood at a time can assist in improving energy efficiency of reprogramming for sensor networks.

We consider two reprogramming models. In the first model, one (or a few) sensor with the new program is dropped to the sensor field. This sensor then works as the base station, and broadcasts the new program to all the sensors in the network via radio. We propose a multihop reprogramming protocol, MNP [33], for this model. MNP provides a reliable and energy efficient service to propagate new program code to all the sensors. To reduce message collision, MNP uses a sender selection algorithm that attempts to guarantee that there is one sender in a neighborhood at a time. Moreover, the sensors that lose in the sender selection algorithm and are not interested in receiving code from its neighbors are put to sleep. In this way, MNP reduces the active radio time of sensors during reprogramming and saves energy. To our knowledge, MNP is the first reprogramming protocol for sensor networks that addresses energy efficiency during reprogramming process and provides quantitative measurements of energy saving. We implement MNP on Mica2 and XSM motes and simulate it on TOSSIM [37], a discrete event simulator designed for TinyOS sensor platform. We will present the evaluation results of both the hardware experiments and simulation.

In the second model, we consider the case where each sensor (or a subset of sensors) has one part of the new program. The sensors communicate with each other to get the remaining parts of the program. We propose *Gappa* [66] for such gossip among sensors. *Gappa* extends the sender selection algorithm from MNP to multiple radio channels. The multi-channel sender selection algorithm tries to guarantee that on each channel, only one sender transmits code in a neighborhood at a time. If a sensor loses in the sender selection on one channel, it will compete to transmit on a different channel that is available. We propose two types of channel allocation schemes. Fixed channel allocation scheme assigns one channel to each segment, while variable channel allocation scheme allows sensors to randomly select a channel from all the available channels. We implement both schemes, and show that the variable channel allocation scheme enables higher concurrency during reprogramming, and hence, performs better. Similar to MNP, in *Gappa*, the sensors that are unable to transmit on any channel and are not interested in receiving code from their neighbors are put to sleep.

Moreover, observe that all the existing protocols on network reprogramming [12, 23, 30, 47, 60], including MNP and *Gappa*, use automatic repeat request (ARQ) scheme to recover from packet losses. We propose a new reliability scheme [65] which is a hybrid approach of forward error correction (FEC) and ARQ. We perform a case study on MNP, and study the effect of adding two different FEC codes: simple XOR code and Reed-Solomon (RS) code, to MNP. We show the improvement on reprogramming speed and reduction on energy consumption.

We also study the issue of providing authentication for reprogramming, and more generally, bulk data dissemination. We propose a symmetric key based authentication protocol for reprogramming. This protocol is based on the secret instantiation algorithm from [18, 31], which requires only $O(\log n)$ keys to be maintained at each sensor. We categorize the adversaries into two types, mote-class adversaries and laptop-class adversaries. A mote-class adversary has limited energy, and cannot use extensive denial of service attacks. A laptop-class adversary is more powerful and can launch denial of service attacks. For the case where only mote-class adversaries exist (e.g., a testbed environment), we propose a simple algorithm [68] and show that it authenticates reprogramming process with low cost. For the case where laptop-class adversaries exist, we develop a mechanism that allows sensors to authenticate the data they receive *before* they store it to EEPROM (an energy consuming operation). In this way, the denial of service attacks from laptop-class adversaries are mitigated. We consider the problem of bulk data dissemination in three scenarios: multihop dissemination of large amount of data (e.g., reprogramming), multihop dissemination of moderate amount of data (e.g., network monitoring, difference-based reprogramming), and single-hop dissemination (e.g., communication within a cluster or a single-hop network). We show that using symmetric key based authentication can significantly reduce the cost of secure dissemination of a moderate amount of data compared to use of public keys, especially in the second and third scenarios. We also propose additional schemes to reduce the cost of secure data dissemination in the first scenario, and show the effectiveness of these schemes. We show the applicability of our protocol by integrating it to the existing reprogramming protocols, including MNP and Infuse [30], in the scenarios mentioned above.

1.6 Outline of The Dissertation

The rest of the dissertation is organized as follows. In Chapter 2, we present our multihop reprogramming protocol MNP, which is designed for the case where one or a few sensors have the entire new program initially, and they communicate the new program to all the other sensors on a shared channel. We will describe the protocol design, implementation issues, and evaluation results, which include experiments on the hardware Mica2/XSM motes and simulation on TOSSIM.

In Chapter 3, we present *Gappa*, a gossip based multi-channel reprogramming protocol designed for the scenario where each sensor (or a subset of sensors) has one part of the new program initially and communicates with each other to get remaining parts of the program. We will describe two variations of *Gappa*, the one with fixed channel allocation (*fc-Gappa*) and the one with variable channel allocation (*vc-Gappa*), and compare their performance. Energy efficiency is emphasized in both MNP and *Gappa*, as we will show in protocol description and performance evaluation.

In Chapters 4 and 5, we focus on improving the reprogramming service from two aspects: reliable communication and security. In Chapter 4, we propose to add FEC to the ARQ based reliability approaches used in all existing reprogramming protocols. Through a case study on MNP, we show the effects of using FEC codes in reprogramming, and the tradeoff between computation and performance: if more computation resources are available, a more powerful coding scheme can be used to achieve better performance. In Chapter 5, we propose a symmetric key based protocol for authenticating reprogramming process. We consider two cases. In the first case, only mote-class adversaries exist. In the second case, laptop-class adversaries also exist. We show that using symmetric keys can reduce the authentication cost significantly compared to the case where a public key based scheme is used. We also illustrate the applicability of our authentication protocol in other data dissemination scenarios.

In Chapter 6, we survey related work on reprogramming protocols, FEC coding schemes and secure reprogramming for sensor networks. In Chapter 7, we conclude the dissertation and propose the scope of future research.

Chapter 2

MNP: Multihop Reprogramming Protocol for Sensor Networks

In this chapter, we propose MNP, a multihop reprogramming protocol designed for Mica2/XSM motes. MNP provides a reliable and energy efficient service to propagate a new program from one (or a few) sensor to all sensors in the network over wireless radio. As we have pointed out in Section 1.3, one of the problems in reprogramming is the issue of message collision. To reduce the problem of collision, we propose a sender selection algorithm, in which source nodes compete with each other based on the number of distinct requests they receive. The sender selection algorithm attempts to guarantee that in a given neighborhood there is at most one source transmitting the program at a time. Further, our sender selection is greedy in that it tries to select the sender that is expected to have the most impact. Through simulation on TOSSIM, we will show that this simple greedy approach is effective in reducing the concurrent sender problem.

MNP uses pipelining to enable fast data propagation. Through simulation, we will show the effectiveness of pipelining in large scale networks.

MNP is energy efficient because it reduces the active radio time of a sensor node by putting the node into "sleep" state when its neighbors are transmitting a segment that is not of interest. We call this type of sleep contention sleep. To further reduce the energy consumption, we add noreq sleep, according to which, a sensor node goes to sleep if none of its neighbors is interested in receiving the segment it is advertising. Noreq sleep happens at the end of reprogramming, when a node observes that most or all of its neighbors have received the new program. We also introduce an optional init sleep to reduce the energy consumption in the initial phase of reprogramming.

We implemented MNP on TinyOS Mica2 and XSM mote platforms, evaluate its performance through experiments on Mica2 motes, and investigate the performance of MNP in different network settings through simulation on TOSSIM. In our simulation study of MNP, our goal is two-fold. One goal is to study the tradeoff between different metrics, e.g., completion time, energy usage, based on the choices made during reprogramming. Towards this end, MNP provides several tunable parameters. We study how these parameters affect the completion time and energy usage. The second goal is to demonstrate effectiveness of MNP itself. Towards this end, we simulate MNP in different settings and compare it with Deluge [23]. We also show that even if the tunable parameters are chosen suboptimally, the designer can still obtain a considerable energy savings. We also show parameters in MNP can be dynamically adapted (based on the observations from the network during reprogramming) while preserving a large portion of energy savings. Furthermore, when network characteristics are known in advance, the designer can utilize that information to minimize energy usage (respectively, completion time) by choosing appropriate parameter values.

In Section 2.1, we present our code dissemination protocol, MNP. We focus on the sender selection algorithm, reliability issues, and sleep conditions. In Section 2.2, we evaluate the performance of MNP under different network settings. We summarize this chapter in Section 2.4.

2.1 MNP: Protocol Description

In this section, we present the code dissemination protocol MNP. In Section 2.1.1, we present the sender selection algorithm, which is the core of MNP. In Section 2.1.2, we describe the sender-receiver behavior when a node is forwarding code to its neighbors. In Section 2.1.3, we address the reliability issues, including loss detection and recovery. In Section 2.1.4, we describe the operation of the protocol as a state machine. In Section 2.1.5, we discuss the problem when the sensor nodes should reboot with the received program. In Section 2.1.6, we analyze some optimization approaches to save energy. We focus on various situations in which the sensor nodes can be put to sleep.

2.1.1 Sender Selection Algorithm

To simplify presentation, we first consider the case where the program has only one segment. In Section 2.1.1, we present the basic sender selection algorithm for single-segment programs. In Section 2.1.1, we revise this algorithm so that it can be used with programs that have multiple segments.

Basic Sender Selection algorithm

Before describing the algorithm in detail, we illustrate it using an example. Towards this end, consider the sensor network in Figure 2.1. Suppose A transmits the code first and nodes B, C, D, E, and G receive this code. Now, these nodes should not transmit simultaneously as it will cause significant collisions. Moreover, the choice of the sensor that transmits next is not uniform. For example, G is a better choice than D; some of the nodes that D can send data to have already received the data from A.

In our algorithm, each source node maintains a variable *ReqCtr* that indicates the number of distinct requests (from different requesters) it has received so far. *ReqCtr*

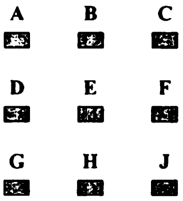


Figure 2.1: Example sensor network

is set to zero when a source starts advertising, and incremented by one every time it receives a *request* that is *destined* to it from a "new" requester. (In order to decide if a node is "new", a node maintains a neighbor table that contains the lower byte of the node IDs it has heard. The size of the neighbor table is restricted to 16 bytes, hence, it can store up to 16 IDs (only the lower byte). If a node ID is not found in the neighbor table, it is considered "new".)

We use two types of messages for sender selection: *advertisement* and *request.* An *advertisement* message has information about the new program (program ID and size) and the source node (source ID and *ReqCtr* value). It has two goals: announcing the arrival of new program, and preventing the source nodes that have fewer requesters from becoming a sender.

When a node, say j, receives an advertisement message from a source node, say k, if j needs the new code, then it sends a request to k. The request sent by j also contains the value of *ReqCtr* that k sent in the advertisement message. While the request is intended (*destined*) for k, it is sent as a broadcast message with k as one of the fields. Thus, when another node, say l, receives the request, l is aware of the fact that k is a potential source. This allows us to account for hidden terminal effect where l could not have received the advertisement message from k. Moreover, by including the value of *ReqCtr* in request message, we allow l to be aware of the number of requests to k. Hence, l can utilize this information to determine who should transmit the code first.

We note that a node sends a request to all senders that send the advertisement messages. This ensures that a node is aware of all the requesters who are likely to receive the code if it is chosen to transmit the code. However, if a node, say k, loses to node l that has more requesters, then whenever k attempts to advertise again (e.g., after l has transmitted the code), k must reset its *ReqCtr* to zero, and recalculate its requesters. This is due to the fact that some of old requesters of k may have already received the code from l.

Based on the above discussion, we describe our sender selection algorithm as two parts: source part and requester part.

Tasks for source. In Figure 2.2, we present the tasks that a source node performs in the sender selection process. This part contains the basic control logic and the actions in response to the received messages.

A source node S broadcasts an *advertisement* message every random interval (we use random interval to avoid message collision). Every time S receives a *request* message, it checks to see if this message is *destined* to it. If the message is destined to it, and is from a "new" requester that S has not seen before, S increments *ReqCtr* by one. If the *request* message is destined to some other node and that node has a higher *ReqCtr* value, then S stops advertising and goes to *sleep* state.

When S overhears an *advertisement* message from another source node, it compares the *ReqCtr* value of that node with its own. If the other node has more requesters than S does, it gives up advertising and goes to *sleep* state. (Note that this cannot cause deadlock, as the node with the highest *ReqCtr* - with appropriate tie breaker on node ID - will succeed.)

If S receives a "StartDownload" message or a data packet, i.e., some node in the neighborhood has won this round of sender competition, S stops advertising and goes to *sleep* state.

For a "sleeping" sensor node, nothing is active except a timer. When the "sleep" timer fires, the source node wakes up, and listens to radio for a short amount of time (e.g., 0.5 second) before it restarts advertising. During the listen interval, the node could be put back to sleep if it finds that a neighboring node is transmitting or is going to transmit a program (i.e., *ReqCtr* greater than 0) that it is not interested in.

Contention sleep. We have seen that a node goes to sleep when it loses in the sender selection algorithm or detects ongoing traffic. In this case, sleep is caused by contention for the shared wireless channel. We call this type of sleep, *contention* sleep. The sleep action is triggered by the arrival of a message (an advertisement, a request, or a data message). Due to the broadcast nature of wireless medium, a message is received by all the sensor nodes that are within its transmission range. These sensor nodes are put to sleep simultaneously and wake up at approximately the same time to join the sender selection. The sleep duration should be determined based on the expected transmission period and the network density.

The advertising phase ends when a source node has sent a given number of advertisements successively (without "sleeping"). At this point, if it has received one or more requests, it will become a sender and start transmitting code. Otherwise, it will wait for t seconds, then restart advertising. t is set to t_l initially, every time the node fails to receive any request, the wait period t doubles. Once t reaches its upper bound t_h, it stays constant.

Noreq sleep. When a source node is in wait period, it can turn its radio off and go to sleep. We call this type of sleep *noreq* sleep. A source node goes to *noreq* sleep when it observes that none of its neighbors is interested in the data it is advertising. *Noreq* sleep is needed for energy efficiency, especially at the end of reprogramming. Consider the scenario when all (or most) sensor nodes in the network have got the new program, each of them advertises infrequently. With *noreq* sleep, these nodes go to sleep when they are not advertising; while without *noreq* sleep, they have to stay awake all the time.

Tasks for requester. In Figure 2.3, we show the tasks that a requester

```
Source: (in advertise state)

Broadcast an advertisement message every random interval
After advertising $N$ times (without sleep):
   if ($my.ReqCtr > 0$)
      Become a sender, and start forwarding code
   else
      Wait for $t$ seconds, then restart advertising
   endif

During advertise interval:
(a)
if a request message ReqMsg arrives
   if ($ReqMsg.DestID == my.ID$)
      if ( IsNew(ReqMsg.SourceID) )
         $my.ReqCtr$ ++
      endif
   else //the message is destined to some other node
      if ($ReqMsg.ReqCtr > 0$) and
      (($ReqMsg.ReqCtr > my.ReqCtr$ ) or
      ($ReqMsg.ReqCtr == my.ReqCtr$) and ($ReqMsg.DestID > my.ID$))
         Go to "sleep" state, $my.ReqCtr = 0$
      endif
   endif
endif

(b)
if an advertisement message AdvMsg arrives
   if ($AdvMsg.ReqCtr > 0$) and
   (($AdvMsg.ReqCtr > my.ReqCtr$) or
   ($AdvMsg.ReqCtr == my.ReqCtr$) and ($AdvMsg.SourceID > my.ID$))
      Go to "sleep" state, $my.ReqCtr = 0$
   endif
endif

(c)
if a "StartDownload" message or a data packet arrives
   Go to "sleep" state, $my.ReqCtr = 0$
endif
```

Figure 2.2: Tasks of the source in sender selection algorithm

performs in the sender selection process. If a node hears an *advertisement* that announces the availability of a new program, it waits for a short random interval (we use random backoff to prevent response implosion from multiple neighboring nodes). Then, the node broadcasts a *request* message that is destined to the advertising node. As mentioned earlier, it also puts the *ReqCtr* information of that advertising node in the *request* message.

```
Requester:

if an advertisement message AdvMsg arrives
    if it is a "new" program
        Prepare request message ReqMsg:
            ReqMsg.DestID = AdvMsg.SourceID
            ReqMsg.ReqCtr = AdvMsg.ReqCtr
        Send ReqMsg after a short random interval
    endif
endif
```

Figure 2.3: Tasks of the requester in sender selection algorithm

Sender Selection for Programs with Multiple Segments

In order to apply sender selection to the cases where the programs have multiple segments, we make the following changes to the algorithm in Figures 2.2 and 2.3.

1. Each *advertisement/request* message contains an additional field *SegID* (segment ID).

2. When a node receives an *advertisement* message for a segment that it does not have, it sends a *request* that contains the ID of the segment it expects to receive. For example, if the *advertisement* is for segment 3, and the node has received segment 1 in the past, it will request for segment 2.

3. When a node receives a *request* for segment y while advertising segment x, if $y < x$, then it starts advertising segment y. This is true even if the *request* is

not "destined" to this node.

4. When node l receives an *advertisement* for segment y from node k while l is advertising segment x, if $y < x$, and k has already received at least one *request*, then l goes to *sleep* state. (We give higher priority to a lower segment.)

5. Timeouts are used so that a node can determine whether it should advertise the current segment or the next one.

2.1.2 Tasks in Downloading a Segment

When a node decides to become a sender, it broadcasts a "StartDownload" message to announce this fact, and then starts sending code packet by packet. A node will change to *download* state once it hears a "StartDownload" message with expected segment ID. Since a node always receives segments in order, the expected segment ID is the highest segment ID the node has received so far plus one. The node in *download* state sets the sender (the node that has sent the "StartDownload" message) to be its *parent* (for that segment).

We note that although the sender selection algorithm attempts to keep only one active sender in a given neighborhood, it is possible to have multiple active senders due to time-varying link properties. Hence, a node may receive code from multiple senders. In our protocol, we allow a sensor node to receive data packets from its parent as well as other senders, as long as the segment ID is the expected one.

When a node is in *download* state, it receives the data packets and stores them in EEPROM. At the same time, it keeps track of missing packets. The download process ends when the receiver receives an "EndDownload" message from its parent. At this point, if the node has successfully received the whole segment, it will go to *advertise* state. Otherwise, the node goes to *fail* state.

Parent-children relationship is one-directional: the child knows who its parent is. However, the parent does not know who its children are. It is possible that the receiver never gets the "EndDownload" message. The reason can be the sender dies as it is sending packets, or the "EndDownload" messages collide with other messages. To avoid being stuck in *download* state, the node in *download* state sets a timer when it is waiting for the next packet from its parent. It will wait for reasonably long time until it concludes that this download process has failed. Then it goes to *fail* state.

We also notice that although the sender selection algorithm has effectively reduced the hidden terminal problem, the problem still persists to a certain extent. For example, consider a scenario where nodes k and l are out of communication range of each other, but both can send messages to and receive messages from node j. Suppose l wakes up when j is in *download* state, and is receiving data packets from its parent k. Now, l listens to the radio for a while (it cannot hear k's messages), then starts advertising, and will probably transmit data if it gets requests from its neighbors. The advertisements or data packets from l can cause collisions on j with k's transmission. We have tried to deal with this situation by letting the node that is in *download* state (in this case, j) send a "quiet" message, asking its neighbors to keep silent, if it receives advertisements or data packets that are not from its parent. However, we found that the "quiet" messages sent by a node (j) interfere with the transmission from its parent (k), and cause more collisions. Therefore, we allow a certain degree of hidden terminal effect.

2.1.3 Reliability Issues: Loss Detection and Recovery

In MNP, each packet has a unique ID. Each receiver is responsible for detecting its own loss. Since the size of the segment is small and pre-determined, a node maintains a bitmap (which we call *MissingVector*) of the current segment it is receiving in memory. Each bit in *MissingVector* corresponds to a packet. All the bits are initially set to 1. When a node receives a packet for the first time, it stores that packet in

EEPROM and sets the corresponding bit in *MissingVector* to 0. In this way, we guarantee that each packet in a segment is written to EEPROM only once. One additional advantage of this mechanism is that a receiver can track packets, when the packets in a segment are received out of order.

In MNP, each node has a *ForwardVector*, which is a bitmap of the advertised segment, and is an indicator of the packets the node needs to send if it becomes a sender. When a node sends a *request* message, it puts the loss information (its *MissingVector*) in the *request* message. When the advertising node receives the request, it marks its *ForwardVector* according to the loss information. Therefore, the *ForwardVector* of an advertising node is the union of the *MissingVectors* in the *request* messages that the node has received. A node only sends the packets indicated in the *ForwardVector*. We restrict the length of the segment to be no longer than 128 packets, so that the maximal size of *MissingVector* is only 16 bytes, and thus fits into a radio packet.

2.1.4 The Big Picture

In Figure 2.4, we show an overall picture of MNP. MNP operates as a state machine. It includes the following states: *idle*, *download*, *advertise*, *forward*, *sleep* and *fail*. *Fail* state is used to avoid infinite waiting. A node always sets a timer when it is waiting for the next packet or the retransmitted packet from its parent. If it does not receive any packet from its parent when the timer fires, it will temporarily go to *fail* state. A node in *fail* state releases EEPROM resource, and switches to *idle* state immediately.

2.1.5 When to Reboot

When a sensor node receives all the segments of a program, it can reboot with the new program. One way to do it is that, a node reboots only when it receives an external "reboot" signal. The time when the signal is sent is based on empirical data

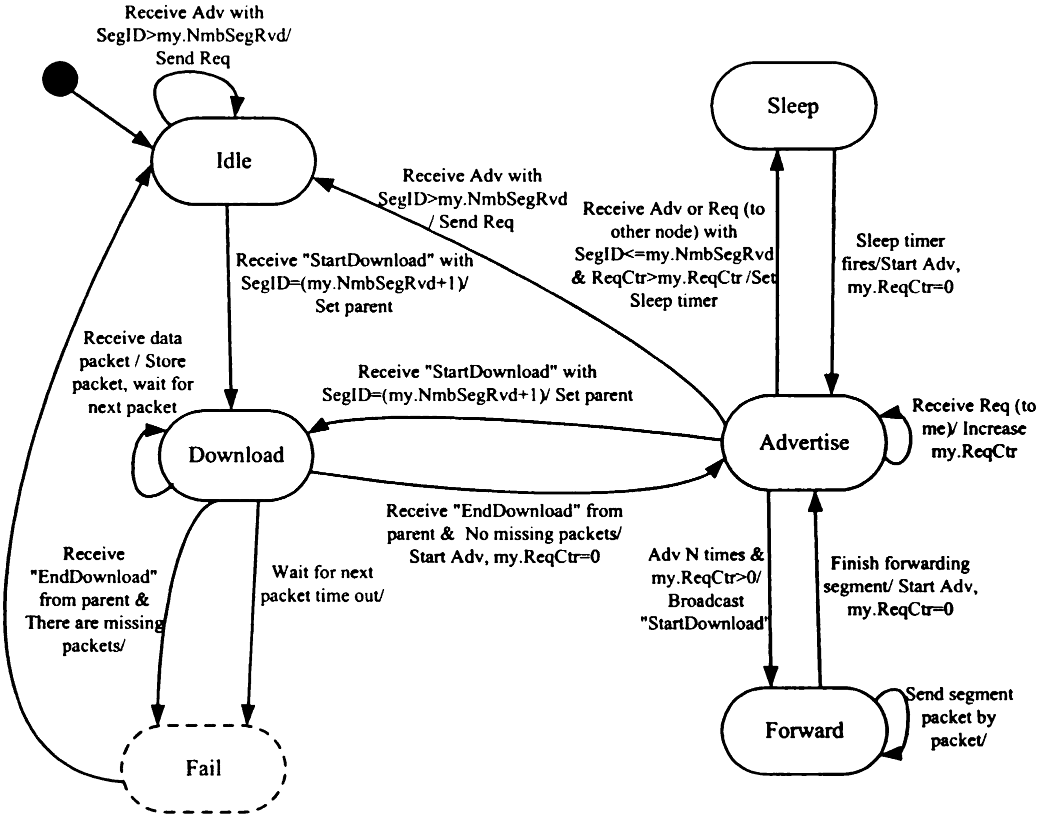

Receive Adv with SegID>my.NmbSegRvd/ Send Req
Idle
Sleep
Receive Adv with SegID>my.NmbSegRvd / Send Req
Receive Adv or Req (to other node) with SegID<=my.NmbSegRvd & ReqCtr>my.ReqCtr /Set Sleep timer
Sleep timer fires/Start Adv, my.ReqCtr=0
Receive "StartDownload" with SegID=(my.NmbSegRvd+1)/ Set parent
Receive data packet / Store packet, wait for next packet
Receive "StartDownload" with SegID=(my.NmbSegRvd+1)/ Set parent
Download
Advertise
Receive Req (to me)/ Increase my.ReqCtr
Receive "EndDownload" from parent & No missing packets/ Start Adv, my.ReqCtr=0
Receive "EndDownload" from parent & There are missing packets/
Wait for next packet time out/
Adv N times & my.ReqCtr>0/ Broadcast "StartDownload"
Finish forwarding segment/ Start Adv, my.ReqCtr=0
Forward
Send segment packet by packet/
Fail

Figure 2.4: MNP: the state machine.

from experiments. Alternatively, reboot can happen automatically. For example, a node reboots with the new program as soon as it receives the entire program. In this case, the new program should include the reprogramming service, so that the node can continue to serve as the source node after reboot. Yet another choice is to let a node decide the time to reboot based on its local estimation of its neighbors. For example, if a source node has sent K (a pre-determined parameter) advertisements of the highest segment ID and receives no request, it assumes that all its neighbors have received the entire program, and reboots itself with the new program. We follow this approach. The actual value of K is decided by the designer, based on the empirical results from experiments and simulation.

2.1.6 Optimization on Energy Conserving

In this section, we propose several possible optimization approaches. In Section 2.1.6, we propose an optional sleep situation, *init* sleep, which can be used to reduce energy consumption in the initial phase of reprogramming. In Section 2.1.6, we refine the *contention* and *noreq* sleep conditions based on our observation.

Adding Init Sleep

Before reprogramming starts, if the application running on the sensors does not require radio communication, we can use some energy conservation approach (such as TDMA [29,63], S-MAC [73]) to turn off the radio when it is not needed. Alternatively, we use a simple approach, called *init* sleep, where nodes take short naps before the "propagation wave" of the new program arrives. Specifically, in the initial state, a sensor node keeps its radio on, and listens to the radio for a short amount of time (e.g., 0.5 second), during which if it does not hear any advertisement, request, or data, it goes to sleep. When the sleep timer fires, it wakes up and listens to the radio again. This "listen-sleep" pattern continues until the node hears something that it is interested in, upon which, it will keep its radio on (Figure 2.5).

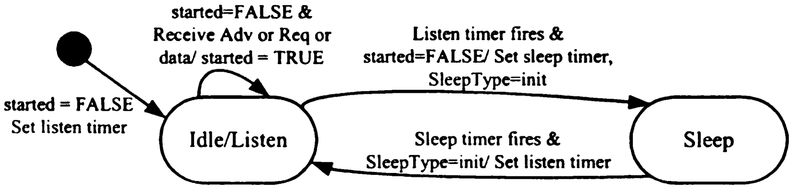


Figure 2.5: Init Sleep (Listen state is the same as Idle state). *started* is set to FALSE initially. If an advertisement, request, or data is received, *started* is set to TRUE. The radio is kept on after *started* becomes TRUE.

The purpose of *init* sleep is to reduce idle listening in the initial phase of reprogramming. Init sleep is optional, and can be replaced by any other energy conservation approach. It can also be disabled if the underlying application running on the sensor nodes (the "old" application) requires that radio is on.

Refined Contention and Noreq Sleep Conditions

Putting a node to sleep in the three situations (contention sleep, init sleep, noreq sleep) saves energy and reduces overhearing. However, if a node sleeps for too long, reprogramming process will be delayed due to the fact that the sleeping node misses many advertisements from its neighbors or it does not send updates to its neighbors in time. When a node meets the conditions of contention sleep (loses in the sender selection algorithm or detects ongoing transmissions) or noreq sleep (has sent out N advertisements but no request is received), it should backoff for a certain amount of time before it restarts advertising. The backoff period corresponds to the contention sleep period for contention sleep, and noreq wait period t for noreq sleep. During the backoff period, a node decides to turn off its radio and go to sleep, or listen to the channel with its radio on, based on its status and its knowledge about its neighbors. To obtain adequate information of its neighborhood, a node does not sleep through the entire backoff period. Rather, a sleeping node wakes up from time to time to update its environmental knowledge, based on which, it decides its next action.

Short naps before activation. If a node has not received the entire program, it should keep its radio on to wait for advertisements of higher segments. However, this will keep nodes awake most of the time during reprogramming, which results in increased energy consumption. Thus, we use an approach that is similar to init sleep. Before a node gets the entire program, rather than keeping its radio on all the time, it can take short naps, wake up and check the channel from time to time. A node starts the "listen-sleep" pattern when it has just received a whole segment, and has no idea when the next segment will arrive. Once it detects some activity of a higher segment, it will keep its radio on continuously thereafter.

To implement this, a node keeps a boolean variable *activated*, indicating whether it is in active updating phase. When a node receives an entire segment, it sets its *activated* to FALSE. Whenever it receives any messages of higher segments, it sets *activated* to TRUE. When a node meets the condition of *contention* sleep or *noreq* sleep, if it has already received the entire program, it goes to sleep. Otherwise, if *activated* is TRUE, it keeps radio on and listens to channel; if *activated* is FALSE, it takes short naps, wakes up and checks the radio from time to time during the contention or noreq sleep period. Once *activated* becomes TRUE, it enters active updating phase and keeps awake from then on. If we set *activated* to FALSE in the initial state, init sleep can be considered as a special case of this activating approach.

Adjusting noreq wait period t. In Section 2.1.1, we have mentioned that if a node fails to receive any request continuously, its noreq wait period t increases exponentially from t_l to t_h. We identify two situations in which t is reset to t_l.

First, if during the advertise and wait period, a node receives advertisements, requests, or data messages that take it to a different state (download, forward, sleep, or idle), when it returns to the advertise state, it will reset the wait period to t_l. Second, if the source node detects a *potential* requester, it will set t to t_l. Specifically, a source node maintains a variable *LowCtr*, which is the number of advertisements and requests (including those that are not destined to it) that contain a lower segment ID (the segment ID is lower than the highest segment ID this node has received) it has heard so far. *LowCtr* is reset to 0 whenever a node switches from "radio off" to "radio on" or when a node starts advertising. *LowCtr* is checked at the end of each wait period t, before a node restarts advertising. If *LowCtr* is greater than 0, which means that one or more neighbors might need to be updated, then the next wait period t will be reset to t_l.

When a node is in noreq sleep, it wakes up every t_n seconds, listens to the channel for a short period (we use 0.5 second). If the messages received during this interval make *LowCtr* greater than 0, which means that one or more neighbors might need to be updated, then the node will reset the next wait period t to t_l, and start advertising. As a result, the backoff period ends.

Note that when a node is in listen interval, although it does not send advertisements, it still sends requests or turns to download state when necessary. If a node receives an advertisement that contains the segment that it is interested in, it will send requests to the advertising node. If a node receives a "StartDownload" message or data message, and the node is interested in receiving the segment, it will turn to download state. In this case, the backoff period ends.

2.2 Evaluation Results

Our target platform is TinyOS Mica2/XSM motes, with 433MHz radio. A Mica2/XSM mote has 128KB of program memory, 4KB of RAM, a 7MHz 8-bit microcontroller, and 512KB external flash storage (EEPROM).

We fully implemented MNP on Mica2 and XSM mote platforms, and used two methods to evaluate the behavior of MNP. The first method is to run the code on TinyOS hardware, Mica2 motes. We experimented in a classroom and on a grass field in a grid topology. The purpose of these experiments is to verify the correctness of the algorithm and observe the effectiveness of the sender selection protocol. Due to the limitation on the number of available motes and the space to perform experiments, we were unable to experiment with networks of large scale. Therefore, the second method is to use TOSSIM. TOSSIM is a discrete event simulator for TinyOS wireless sensor networks. We use TOSSIM to investigate the behavior of MNP when it is applied to a large network.

In the rest of this section, we first present the indoor and outdoor experiment results with Mica2 motes. These results are based on the basic version of MNP without pipelining. We did not use pipelining because the number of motes and the space for performing the experiments were relatively small, and pipelining would be significantly helpful only when the network is large and several non-overlapping communication cells exist. In the second part, we present simulation results using TOSSIM.

2.2.1 Experiments with Mica2 Motes

TinyOS allows developers to specify the power level a Mica2 mote uses for its radio communication. The range of power level is from 1 to 255. In our indoor experiments, we use the lowest power levels (1 and 2). In outdoor experiments, we use power level 10 and default power level (255).

In these experiments, we place sensor nodes in a grid. The base station, which has the new program image, is always put in the upper-left corner of the grid. We expect that these results would be valid if the number of sensors is increased 4 times and the base station is kept at the center.

We tested our algorithm in both indoor and outdoor environments. Due to limitation of space for performing experiments, we fixed the inter-node distance, i.e., the distance between two neighboring nodes, to 8 feet. We repeated our experiments under the same setting with different power levels. By using different power levels, we change the communication range of sensors, and thus the number of hops to propagate the program through the network.

The goal of the experiments is to examine the behavior of our sender selection protocol. For this purpose, each node records the time when it gets the full program image ("get code time") and the ID of its parent (parent ID). Further, we synchronize all the nodes before the experiment starts, so that the time reported by each node is consistent. Note that this synchronization is not used by the algorithm. Rather, it is used to collect data consistently.

Indoor Experiments

We deployed 25 sensor nodes in a classroom area (approximately 32' by 32'), in a 5 by 5 grid. In order to see multi-hop effect, we chose the lowest power levels: power level 2 and power level 1.

Figure 2.6(a) shows the parent-children relationship of the experiment with power level 2. Each grey dot represents a sensor node. From each node, there is an arrow pointing to its parent. According to the "get code time" value and parent ID, reported by each sensor, we can compute the order of sensors becoming senders, which is marked on the figure. As we can see in Figure 2.6(a), our sender selection protocol worked pretty well, only two nodes, other than the base station, became senders one after another. All other sensors that joined the sender selection were put in "sleep" state.

In Figure 2.6(a), most of the sensors receive code directly from the base station. When we reduce power level to 1, as shown in Figure 2.6(b), more sensors are not covered by the base station, thus have to obtain code from other intermediate nodes.

Outdoor Experiments

We performed two sets of experiments on a grass field. In the first set of experiments, we deployed 49 motes in a 7 by 7 grid, in a 48' by 48' area. In the second set of

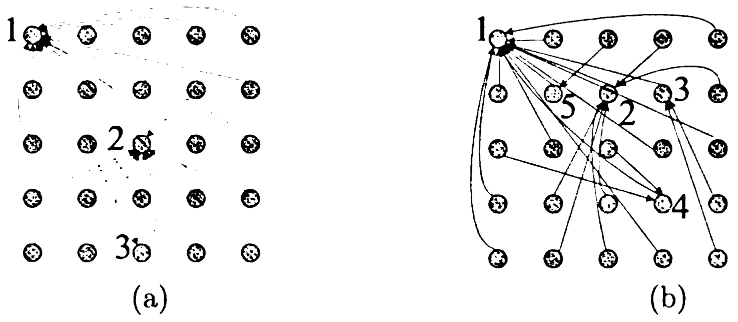
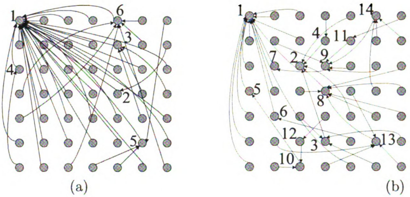


Figure 2.6: Indoor experiments for 5 by 5 grid with (a) power level = 2, time = 5 minutes; (b) power level = 1, time = 8 minutes. Program size: 1500 packets (33KB).

experiments, we placed 50 motes in a 5 by 10 grid, in a 72' by 32' area. The purpose of using this 5 by 10 grid topology is to better examine multi-hop behavior in the code dissemination process. We used two different power levels: full power level (the default value in TinyOS), and power level 10. Figure 2.7 shows the parent-children relationship and the order of source nodes becoming senders for 7 by 7 grid. Figure 2.8 shows the results for 5 by 10 grid.

We notice that the nodes that are away from the base station are more likely to become senders. This is desirable, because these nodes have a larger number of nodes in their neighborhood that are not covered by the base station. As shown in Figure 2.6, 2.7 and 2.8, when nodes are working at a lower power level, more nodes become senders, and each sender has a smaller group of followers. Therefore, more hops are involved in propagating code to the nodes that are far away from the base station. In our experiments, we did not observe the situation where two nearby nodes were transmitting simultaneously. This shows that the sender selection algorithm, although imperfect, achieves its goal of selecting a sender with the largest impact and selecting at most one sender in a neighborhood.

We repeated our experiments several times. We found that the results are sim-



(a)

(b)

Figure 2.7: Outdoor experiments for 7 by 7 grid with (a) full power level, time = 25 minutes; (b) power level = 10, time = 35 minutes. Program size: 1500 packets (33KB).

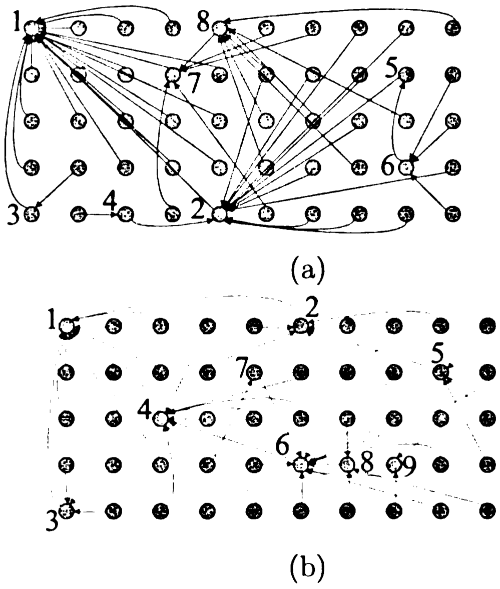


Figure 2.8: Outdoor experiments for 5 by 10 grid with (a) full power level, time = 35 minutes; (b) power level = 10, time = 45 minutes. Program size: 1500 packets (33KB).

ilar. Although the actual sensor nodes that became sources differed from one run to another, the sender selection algorithm ensured that two nearby sensors never transmitted simultaneously. Moreover, in these experiments, the sender selection algorithm selected nodes that were farther from the base station (respectively, previous sources).

2.2.2 Simulation Results

In TOSSIM, the network is modelled as a directed graph. Each vertex in the graph is a sensor node. Each edge has a bit-error rate, representing the probability with which a bit can be corrupted if it is sent along this link. Asymmetric links exist in this model since the bit-error rate for each edge is decided independently. We decide the bit-error rate based on our experience with Mica2 motes. Specifically, the packet loss rate on a one-hop (10 feet) link is around 5%. The loss rate increases with distance, and after 50 feet, the loss rate is 100%.

Since TOSSIM does not model energy consumption, we calculate the energy consumption by counting the operations performed during reprogramming. (Alternatively, we can also use PowerTossim [58] to evaluate power consumption. However, since each simulation lasts for tens of hours, the trace file generated during simulation, which is required by PowerTossim in order to compute the energy usage, becomes too large (of the order of several gigabytes) to process.)

According to the data from Table 1.3, we use Equation 2.1 to compute the energy consumption E (in joules), which is the product of charge Q (in coulombs, 1 nAh is the same as 0.0036 coulombs) and voltage V (in volts).

$$\begin{aligned} E &= Q \cdot V \\ &= 0.0036 \cdot (20 \cdot n_{send} + 8 \cdot n_{receive} + 1250 \cdot t_{idle} \\ &\quad +1.111 \cdot n_{read} + 83.333 \cdot n_{write}) \cdot 3 \\ &= 0.0108 \cdot (20 \cdot n_{send} + 8 \cdot n_{receive} + 1250 \cdot t_{idle} \\ &\quad +2.222 \cdot n_{senddata} + 83.333 \cdot 2 \cdot n_{storedata}) \end{aligned} \tag{2.1}$$

In the above equation, n_{send} and $n_{receive}$ are the number of packets transmitted and received respectively during reprogramming, t_{idle} is a node's active radio time (in seconds), n_{read} and n_{write} are the number of reads and writes respectively executed by EEPROM. EEPROM is read and written in 16-byte blocks (lines). Hence, as each packet has 22 bytes data payload, each *data* packet transmitted involves 2 EEPROM reads, and each data packet stored corresponds to 2 EEPROM writes, i.e., $n_{read} = 2 \cdot n_{senddata}$, $n_{write} = 2 \cdot n_{storedata}$, where $n_{senddata}$ is the number of *data* packets transmitted, $n_{storedata}$ is the number of data packets that are stored in EEPROM.

Equation 2.1 shows that energy consumption is decided by idle listening time

(t_{idle}), message transmissions (n_{send}, $n_{senddata}$) and receptions ($n_{receive}$), and the number of data packets stored ($n_{storedata}$). Among these, $n_{storedata}$ is decided by the size of program (divided in packets) to be transmitted, because our algorithm guarantees that each packet is written to EEPROM only once. Therefore, the key to reducing energy consumption is to reduce idle listening time and message transmissions and receptions. Among these, idle listening time (or active radio time) is the most important factor that affects the energy consumption.

In the current implementation, each segment has 128 data packets. Unless stated otherwise, t_l and t_h are 16 seconds and 512 seconds respectively, init sleep period (and the short sleep period when a node is not *activated*) is 4 seconds, the wakeup interval when a node is in noreq sleep t_c is 4 seconds. The simulations are performed in a grid topology. Due to the fact that the execution time of each simulation is of order of tens of hours, we do not provide confidence intervals.

In Section 2.2.2, we evaluate the effect of the three sleep types in MNP: contention sleep, init sleep, and noreq sleep. In Section 2.2.2, we show the performance of MNP in different network settings. In Section 2.2.2, we further discuss the problem of choosing appropriate contention sleep period.

Effect of Different Sleeps in MNP

We will first show the effect of varying contention sleep period under different network settings. The goal is to identify the guidelines for choosing a "good" contention sleep period under a given network setting. Then we show how adding init sleep and noreq sleep improves performance, especially, conserves energy. These sleep types are in effect at different phases of reprogramming, i.e., contention sleep is effective *during* reprogramming, init sleep is effective at the beginning of reprogramming, and noreq sleep has effect *at the end of* reprogramming (and in maintenance phase). Hence, they are independent, and we can study them one after another.

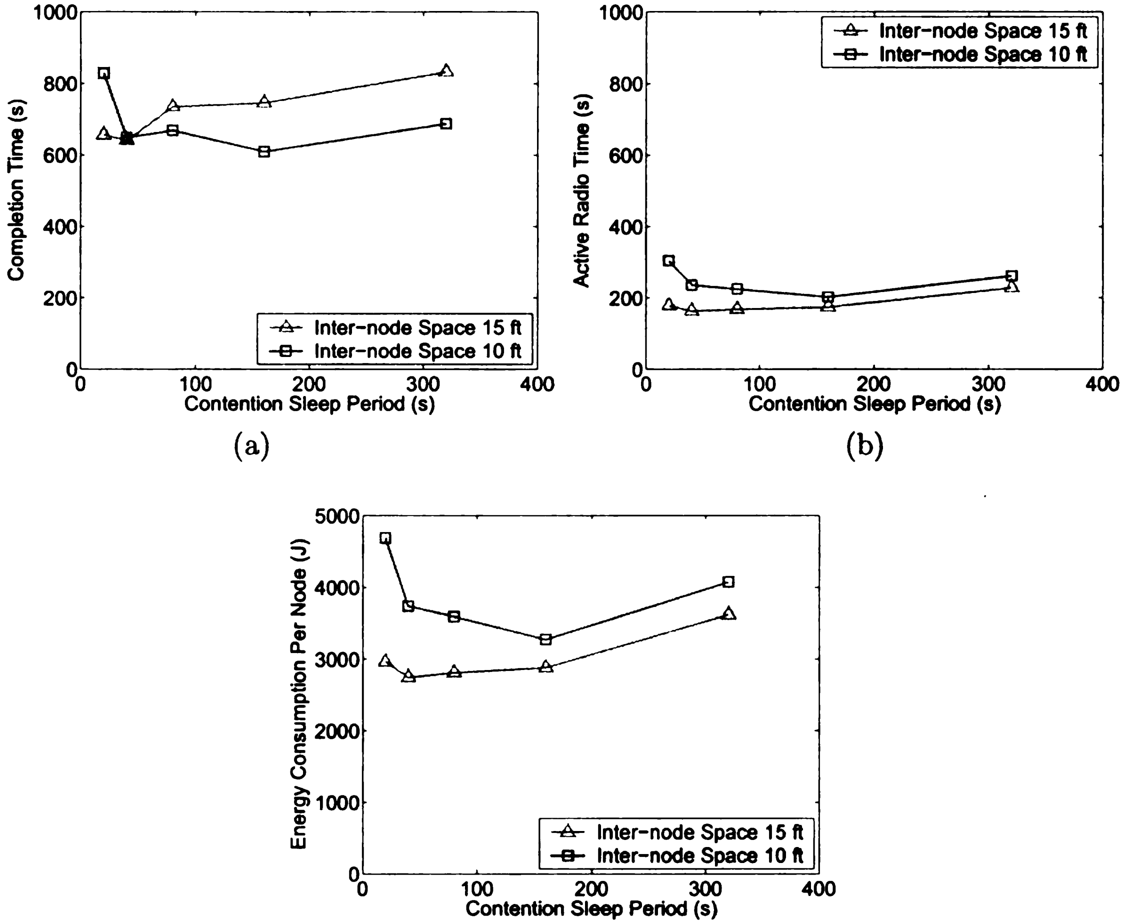
Effect of Contention Sleep.

Varying Network Densities. We conducted the simulations under two network densities, dense network where inter-node distance is 10 feet, and sparse network where inter-node distance is 15 feet. The simulation was done in a 20x20 network. The base station, which has the new program initially, is placed at the corner of the network. The program size is 2 segments (256 packets). We show the effect of varying contention sleep period from 20 to 320 seconds in Figure 2.9. (Images in this dissertation are presented in color.)

We note several things in Figure 2.9. First, for both densities, with the increment of the contention sleep period, the completion time, active radio time and energy consumption drop at the beginning, when the contention sleep period is greater than a certain point, the time and energy starts increasing. The reason to this phenomenon is that increasing contention sleep period essentially reduces nodes' frequency of advertising during reprogramming, thus reduces contention. However, if the contention sleep period is too long, delay increases. Hence, finding the optimal contention sleep period is to find the balance point of reducing contention and not increasing delay.

Second, the turning points in a dense network and in a sparse network are different. The optimal point for contention sleep period appears earlier in a sparse network (in this case, 40 seconds) than in a dense network (correspondingly, 160 seconds). As shown in Figure 2.9(a), when the contention sleep period is less than 40 seconds, data is propagated faster in a sparse network than in a dense network; as we increase the contention sleep period, the dense network starts performing better. This is because in a dense network where each node has a large neighbor set, reducing message collision is a major task, which can be achieved by using a large contention sleep period. On the other hand, in a sparse network, message collision is less. Hence, a long sleep period is disadvantageous.

Third, the active radio time and energy consumption is consistently lower in a



(a) (b)

(c)

Figure 2.9: Completion time, average active radio time per node and average energy consumption per node at different contention sleep periods (20s, 40s, 80s, 160s, 320s) for the dense network (inter-node space is 10 feet) and the sparse network (inter-node space is 15 feet). 20x20 network. Program size: 2 segments (256 packets). (a) Completion time vs. contention sleep period (b) Average active radio time per node vs. contention sleep period (c) Average energy consumption per node vs. contention sleep period.

sparse network than in a dense network, as shown in Figure 2.9 (b) and (c). The reason to this phenomenon is that nodes in a dense network have more neighbors and receive more messages, thus they spend more time in *active updating* phase (they are "activated" earlier, but do not necessarily receive code earlier due to message collision), during which their radio is on.

Varying Network Sizes. In Figure 2.10, we show the completion time, average active radio time per node and average energy consumption per node at different contention sleep periods in a $N \times N$ network, where N increases from 10 to 20. We vary the contention sleep period from 20 to 320 seconds. The inter-node distance is 10 feet. We found that in 10 × 10 and 12 × 12 networks, when the contention sleep period is 40 seconds, the completion time, active radio time and energy consumption are the lowest. As we increased the network size to 14 × 14, 16 × 16, 18 × 18, the lowest point of completion time, active radio time and energy consumption appears when contention sleep period is either 80 seconds or 160 seconds (the performance does not differ much when the contention sleep period is 80 seconds or 160 seconds). When we further increased the network size to 20 × 20, the optimal point of contention sleep period is 160 seconds.

We have already noted that when network density is reduced, the sleep period should be reduced accordingly. Now, we found that the choice of sleep period also depends on the size of the network.

To see why a large sleep period does not work well in a small network, we show the segment completion sequence (in a 10-second window) of nodes in a 10x10 network, when contention sleep period changes from 20 to 320 seconds, in Figure 2.11. We notice that as the contention sleep period increases, the overlapping area of the two segments (which indicates the degree of pipelining) shrinks, then disappears, and at some point, a gap (during which no data is transmitted), between the completion sequences of the two segments, appears and expands. When the contention sleep

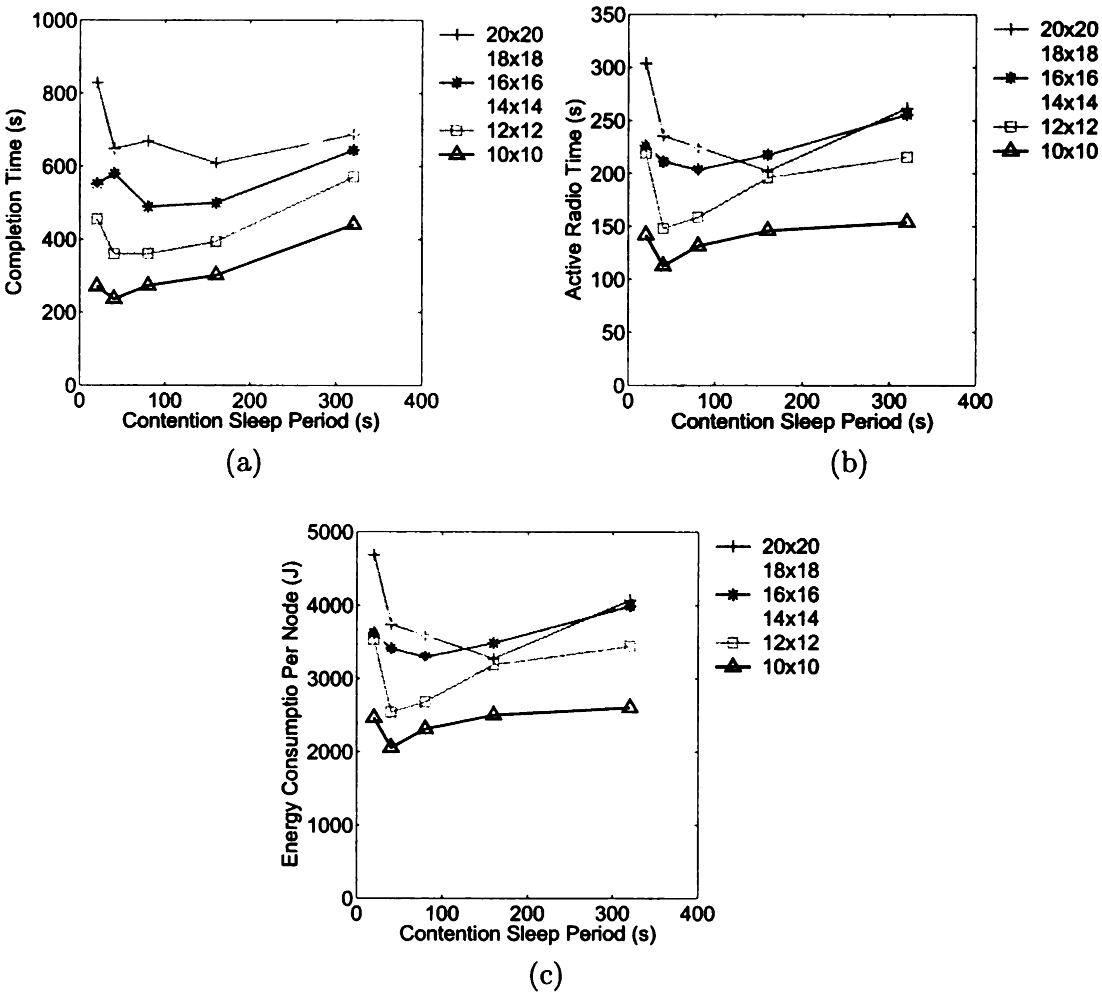


Figure 2.10: Completion time, the average active radio time and energy consumption per node at different contention sleep periods (20s, 40s, 80s, 160s, 320s) in a $N \times N$ network ($N = 10, 12, 14, \ldots, 20$). Inter-node distance: 10 feet. Program size: 2 segments. (a) Completion time vs. contention sleep period (b) Average active radio time per node vs. contention sleep period (c) Average energy consumption per node vs. contention sleep period

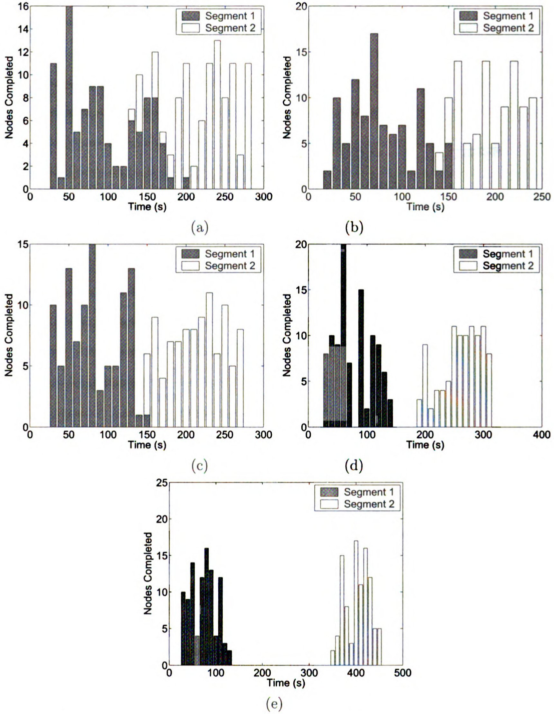
period is large (e.g., 320 seconds, 160 seconds), although propagating each single segment is faster due to less collision, the overall completion time increases because of the gap.

The gap appears when the contention sleep period is longer than the time to propagate *one* segment through the network. The base station is put to sleep (contention sleep) when one of its neighbors starts transmitting code. If the contention sleep period is too long, then the base station is still in sleep state when all the sensor nodes in the network have received the first segment. Because the base station is the only node that has the second segment, all the nodes have to wait until the base station wakes up.

From this example, we conclude that the contention sleep period should be no longer than (usually less than half) the time for propagating one segment through the network. Therefore, the contention sleep period needs to be reduced when the network diameter is reduced. Moreover, noting that the reduction in size of the network essentially reduces the average node density, hence, the conclusion we draw here is consistent with that we had earlier, i.e., contention sleep period should be reduced with the reduction of network density.

In a 10x10 network, when the contention sleep period is 40 seconds, reprogramming consumes the least amount of time and energy, as shown in Figure 2.10 and Figure 2.11. In Figure 2.11(b), we note that the overlapping area between Segment 1 and Segment 2 is very small, which indicates that pipelining does not have much effect when the network is relatively small.

Placing Base Stations. In previous simulations, we kept the base station at one corner of the network. In the following simulations, we test two other situations: the base station is placed in the center of the network, and four base stations are placed at four corners of the network. In Figure 2.12, we compare the completion time, average active radio time per node and average energy consumption per node



(a)

(b)

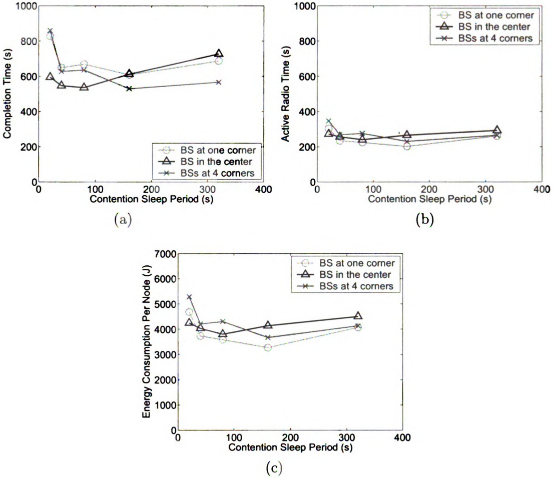
(c)

(d)

(e)

Figure 2.11: The segment completion sequence (in a 10-second window) of nodes in a 10x10 network, when contention sleep period is (a) 20s, (b) 40s, (c) 80s, (d) 160s, (e) 320s. Inter-node distance: 10 feet. Program size: 2 segments (256 packets).

at different contention sleep periods under these three situations.



(a)

(b)

(c)

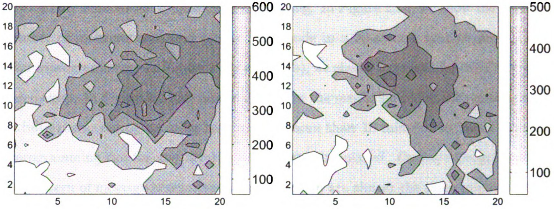
Figure 2.12: Completion time, average active radio time per node and average energy consumption per node at different contention sleep periods under the three situations: one base station is placed at a corner, one base station is placed in the center, four base stations are placed at four corners. 20x20 network. Inter-node distance: 10 feet. Contention sleep period: 160s. Program size: 2 segments. (a) Completion time vs. contention sleep period (b) Average active radio time per node vs. contention sleep period (c) Average energy consumption per node vs. contention sleep period

In Figure 2.12, we show that when the base station(s) are placed at the corner(s), the completion time, active radio time, and energy consumption hit the lowest point when the contention sleep period is 160 seconds; and when the base station is placed in the center, the completion time, active radio time and energy consumption are the lowest when the contention sleep period is 80 seconds. On one hand, placing the base station in the center or placing four base stations at corners effectively reduces the network diameter by half. As we concluded from previous simulations, when the network diameter is reduced, contention sleep period should be reduced accordingly to lessen the effect of latency. On the other hand, the high network density requires a long contention sleep period to reduce message collisions. For example, in the case that four base stations are placed at the corners of the network, significant collisions can occur when the propagation waves meet. Hence, the contention sleep period should be higher than that is used in the small network case (e.g., 40 seconds contention sleep period for a 10x10 network).

Summary on Choosing Contention Sleep Period. We note that long sleep period helps in reducing message collision. We tend to use a long sleep period in a dense network, because reducing message collision is the major task in a dense network. However, a long sleep period also introduces latency. When the network is sparse, the sleep period should be reduced because the latency caused by long sleep periods exceeds the gain from reducing message collision. Also, when the network size is reduced, since it takes less time to propagate a single segment throughout the network, the effect of sleep latency becomes significant. Therefore, choosing an appropriate sleep period is a tradeoff between reducing message collision and reducing sleep latency.

Effect of Init Sleep.

In order to show the effect of init sleep, we tested MNP when it does not apply init sleep, i.e., nodes do not turn their radio off in the initial state. We conducted simulations on a 20x20 network, with 10 feet nodes separation. The program size is 5.6KB (2 segments, 256 packets). Contention sleep period is set to 160 seconds. In Figure 2.13, we compare the active radio time distribution of MNP without and with init sleep. In Figure 2.13(a), we show the active radio time of nodes without init sleep. It shows that the nodes that are far away from the base stations keep their radio on for most of the time during reprogramming. After we add init sleep, as shown in Figure 2.13 (b), this phenomenon disappears, and energy consumption distribution is more even.



(a) (b)

Figure 2.13: Active radio time distribution in a 20x20 network. Contention sleep period: 160s. t_l-t_h: 16s-512s. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). (a) no init sleep (b) init sleep period: 4s

In Table 2.1, we show that, not only the active radio time and energy are reduced (30%), but also reprogramming finishes faster. This is because adding more sleep reduces the number of messages in the network, and hence the number of collisions.

Summary. Adding init sleep to MNP reduces completion time and energy consumption of reprogramming. In the case that the application or lower layer services (e.g., MAC layer protocol) provide energy conservation functionality, we can turn off init sleep.

Table 2.1: Compare MNP without init sleep and with init sleep. Program size: 2 segments (256 packets). Contention sleep period: 160s, $t_l - t_h : 16s - 512s$

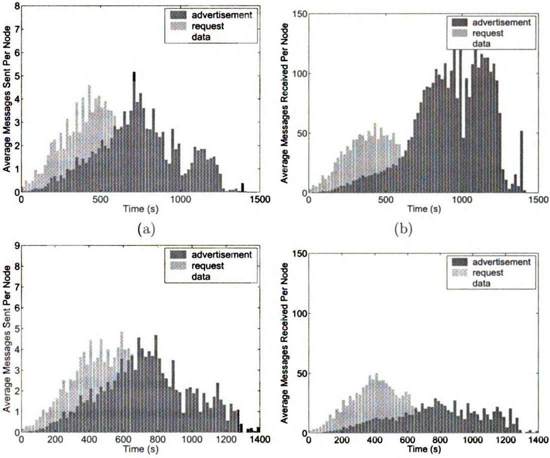
	no init sleep	init sleep 4s
Completion time	655.3 (s)	609.4 (s)
Active radio time /node	290.8 (s)	202.4 (s)
Energy consumption /node	4472.5 (J)	3274.1 (J)
Messages sent /node	96	84
Messages received /node	736	711

Effect of Noreq Sleep.

Now we test MNP when it does not apply noreq sleep, i.e., if a node has sent out a given number of advertisements, but has not received any request, it keeps its radio on and only reduces the frequency of advertising. In Figure 2.14, we show the average number of messages sent and received per node in a 20-second window *during* and *after* reprogramming. In Figure 2.14 (a) and (b), we show the results for MNP without noreq sleep. In Figure 2.14 (c) and (d), we show the results for MNP with noreq sleep. In both cases, reprogramming took a little more than 10 minutes to finish. We let reprogramming run for another 13 minutes after it finished. *During* reprogramming, the pattern of message transmission and reception are almost the same in both cases. Adding noreq sleep does not have much impact *during* reprogramming.

After reprogramming finishes, message transmission increases a little bit, then decreases continuously (Figure 2.14 (a) and (c)). The initial increase is due to the fact that data messages or control messages (advertisements or requests) that indicate current or coming transmissions, which suppress the neighboring nodes from transmitting, no longer exist. In other words, contention sleep no longer exists. After the increase, the nodes still send advertisements, but they are not able to receive any requests, therefore, the noreq wait period grows exponentially, and nodes advertise less and less frequently.

Although the pattern of message transmission does not change when we add noreq sleep, message reception *after* reprogramming (in the maintenance phase) is reduced significantly. As shown in Figure 2.14(b), without noreq sleep, the number of messages (advertisements) received is high after reprogramming. This is because all the nodes in the network are awake. By contrast, with noreq sleep, after the sensor nodes get the program, they sleep most of the time, and only wake up to advertise infrequently to check if there is any node left not reprogrammed. In Figure 2.14(d), we show that with noreq sleep, the number of messages received after reprogramming



(a)

(b)

(c)

(d)

Figure 2.14: The average number of messages sent and received per node in a 20-second window *during* and *after* reprogramming. Program size: 2 segments (256 packets). Contention sleep period: 160s, $t_l - t_h$: $16s - 512s$ (a) Messages sent: without noreq sleep (b) Messages received: without noreq sleep (c) Messages sent: with noreq sleep (d) Messages received: with noreq sleep. Reprogramming completes at around 10 minutes (623 seconds for (a) and (b), 609 seconds for (c) and (d)).

is low (because most of the nodes are sleeping).

In Table 2.2, we compare the performance of MNP with and without noreq sleep. As shown, adding noreq sleep not only conserves energy consumption *after* reprogramming, but also reduces the completion time and energy consumption *during* reprogramming.

Summary. Adding noreq sleep reduces energy consumption at the end of reprogramming since most sensor nodes are put to sleep state when most or all of them have received the new program.

Performance of MNP

In this section, we present the performance of MNP. We first evaluate pipelining and sender selection behavior. Then, we show how the algorithm performs under various network settings, specifically program sizes, network densities, network sizes, and base station positions.

We also compare the performance of MNP with that of Deluge [23]. MNP and Deluge share some common features, such as advertise-request-data handshaking (based on SPIN [28]), dividing a code image into equally sized segments, and pipelining the transfer of segments. MNP uses sender selection algorithm to reduce message collisions. By contrast, Deluge uses Trickle [39] as the suppression mechanism, and it does not turn off nodes' radio during reprogramming. Therefore, for Deluge, a node's

Table 2.2: Compare MNP without noreq sleep and with noreq sleep. Program size: 2 segments (256 packets). Contention sleep period: 160s, $t_l - t_h$: $16s - 512s$, init sleep period: 4s

	without noreq sleep	with noreq sleep
Completion time	623.3 (s)	609.4 (s)
Active radio time /node	302.9 (s)	202.4 (s)
Energy consumption /node	4638.7 (J)	3274.1 (J)
Messages sent /node	91	84
Messages received /node	782	711

idle listening time is the same as the completion time. We simulated Deluge using TOSSIM, under the same network settings as we simulated MNP. (Deluge sets the default segment size to 48 packets. We simulated Deluge when the segment size is set to 96 packets and 128 packets, we found that it performs best when the segment size is 48 packets. Hence, we use 48 packets/segment for Deluge in our simulation.)

Validating Pipelining & Sender Selection.

We observe the pipelining and sender selection behavior of MNP on a 20x20 network, with 10 feet inter-node distance. The size of the program is 5.6KB (2 segments, 256 packets). We set the contention sleep period to 160 seconds. In Figure 2.15, we show the segment completion sequence in a 10-second window. Segment 1 is transmitted first, followed by Segment 2. There is an overlapping area where some nodes start receiving Segment 2, while some are still receiving Segment 1. This overlapping area ends when all the nodes have received Segment 1.

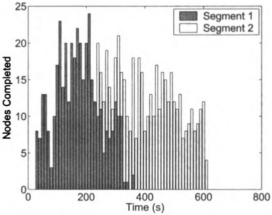
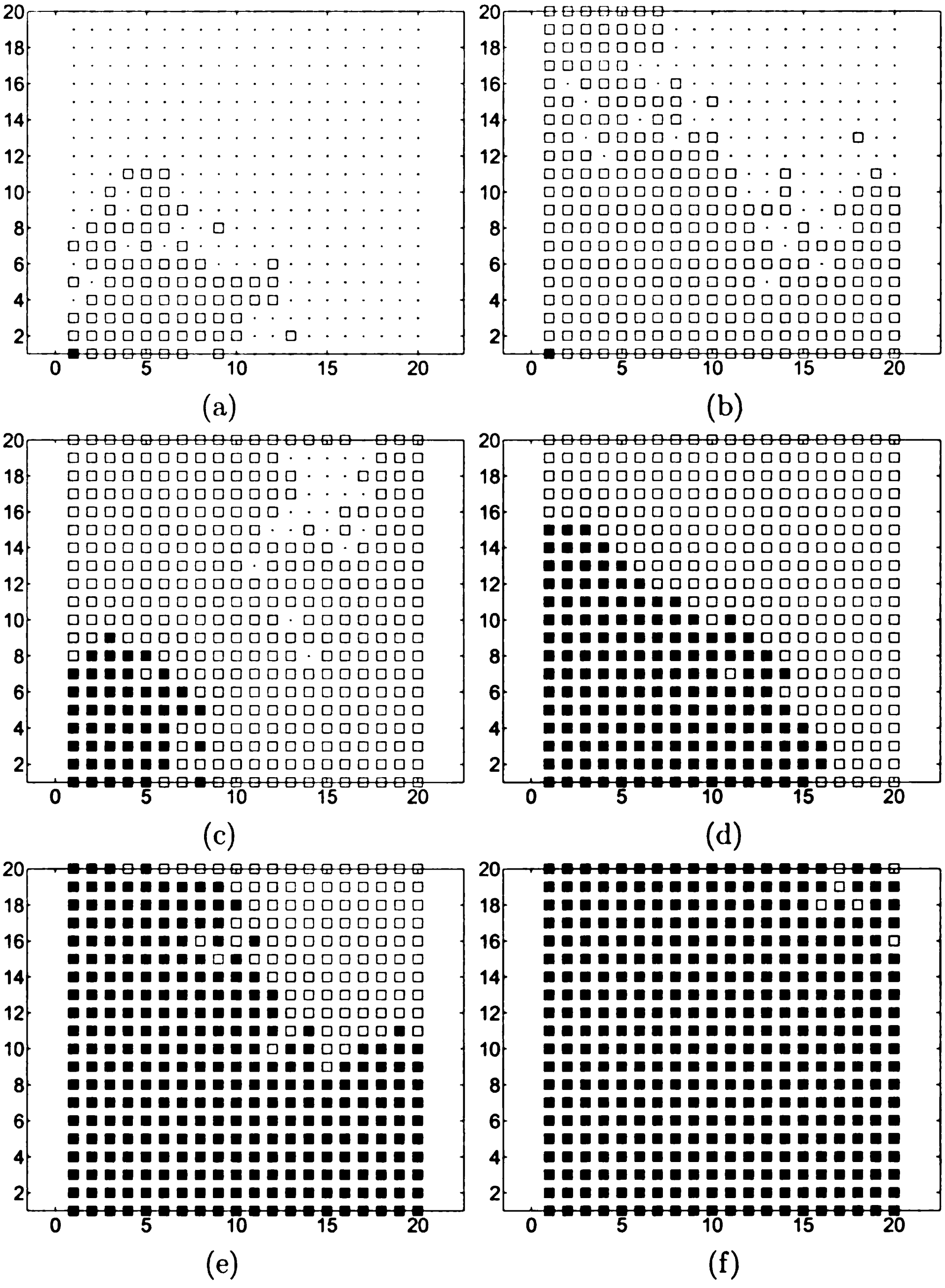


Figure 2.15: The segment completion sequence in a 10-second window. Program size: 2 segments (256 packets). contention sleep period: 160s.

In Figure 2.16, we show the propagation progress for a two-segment program. The segments are propagated gradually from corner to corner. At some point in time, both Segment 1 and Segment 2 are propagating within separate areas of the network (as shown in Figure 2.16(c)).



(a)

(b)

(c)

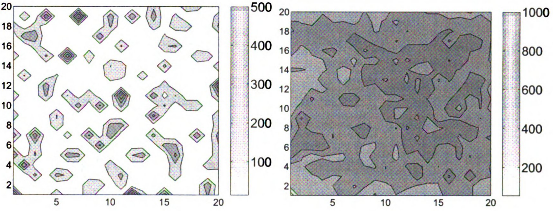
(d)

(e)

(f)

Figure 2.16: Propagation progress for sending two segments in a 20x20 network. The base station is at the left bottom corner. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). Contention sleep period: 160s. Total completion time: 609s. (a) 100s (b) 200s (c) 300s (d) 400s (e) 500s (f) 600s. A node covered by an unfilled square means that it has received Segment 1; a node covered by a filled square means that it has received Segment 2.

In Figure 2.17(a), we show the transmission distribution. We found that the overall message transmission is low. Most nodes send less than 100 messages during reprogramming process. Some nodes transmit more than others. These nodes are *distinct* senders, i.e., they are selected as senders many times. Note that these *distinct* senders are randomly distributed. In Figure 2.17(b), we show that the nodes in the center receive more messages than the ones on the edge or at the corner. Comparing Figure 2.16 and 2.17(b), we found that although the nodes in the center receive more messages than the nodes on the edge, they do not get code earlier due to collisions.



(a) (b)

Figure 2.17: Transmission and reception distribution. 20x20 network. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). Contention sleep period: 160s.

In Figure 2.18, we show the distribution of active radio time. The nodes in the center of the network have higher active radio time than the nodes on the edge. Recall that initially a node takes short naps, and wakes up regularly to check if there are any messages of higher segments. Once it detects any activity that it is interested in, it enters *active updating* phase, during which its radio is continuously on. The active updating phase ends when it has successfully received a whole segment (cf. Section 2.3). This explains why the nodes in the center have higher active radio time. These nodes receive many messages, thus they enter *active updating* phase at an early stage. However, due to message collision, it takes longer for them to receive an entire segment. The nodes in the center spend more time in *active updating* phase, therefore, their active radio time is longer.

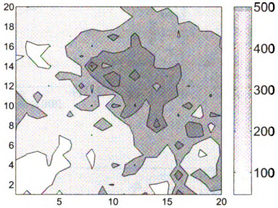
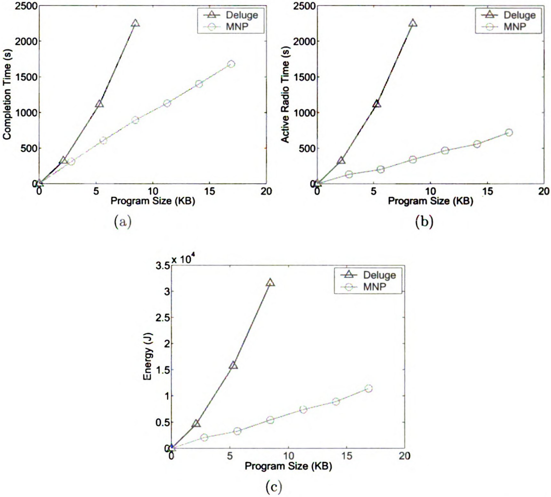


Figure 2.18: The distribution of active radio time. 20x20 network. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). Contention sleep period: 160s.

Varying Program Sizes and Network Densities.

The simulation was done in a 20x20 grid topology, with a base station placed at the corner. We first set the inter-node distance to 10 feet (a dense network). The contention sleep period of MNP is set to 160 seconds. In Figure 2.19, we show the completion time, average active radio time per node, and the average energy consumption per node of MNP and Deluge, under different program sizes. As we can see, for MNP, the completion time is linear with the program size, and the average active radio time per node is about 40% of the completion time. Also, the energy consumption increases linearly, as it is closely related to the active radio time.

We found that MNP outperforms Deluge in both completion time and energy consumption. For example, to propagate a program of 8.45KB (384 packets), the completion time of MNP is 896 seconds, which is only about 40% of the completion time required by Deluge (2247 seconds) when it propagates a program of the same size. Also, the energy consumption of MNP is only 17% of the energy consumption



(a)

(b)

(c)

Figure 2.19: Performance in a dense network. 20x20 network. Inter-node distance: 10 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

of Deluge. In Figure 2.20, we show that MNP has lower transmissions and receptions than Deluge.

We then repeat the same set of simulations on a sparse network, i.e., the inter-node distance is set to 15 feet. The contention sleep period of MNP is set to 40 seconds. We show the corresponding results in Figure 2.21 and 2.22. We found that the completion time of MNP and Deluge are closer compared to the dense network case, while MNP is still up to 30% faster than Deluge (Figure 2.21 (a), when the program size is larger than 14KB). As shown in Figure 2.21 (c), MNP saves up to 75%

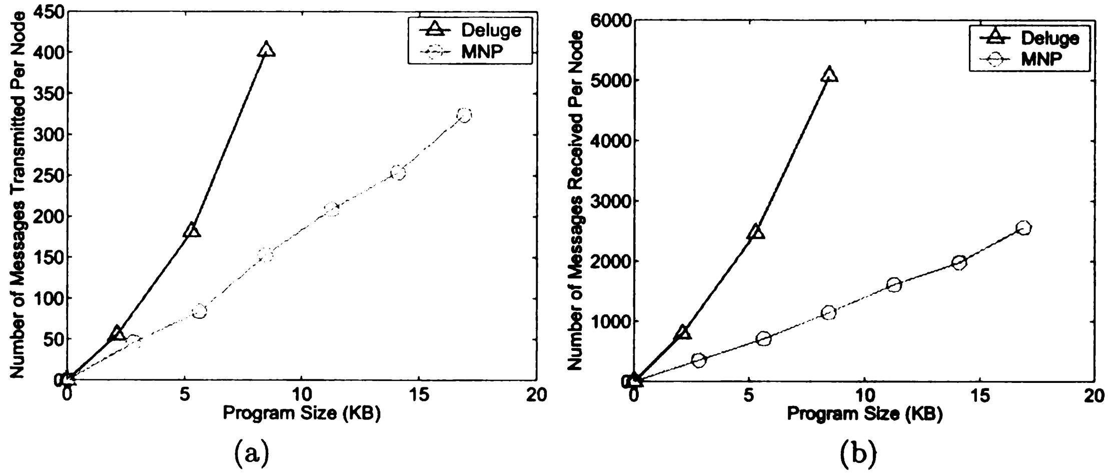


Figure 2.20: Transmissions and receptions in a dense network. 20x20 network. Internode distance: 10 feet. (a) number of messages transmitted per node (b) number of messages received per node.

of energy compared to Deluge. In Figure 2.22, we show that MNP has slightly lower message transmissions compared to Deluge, and up to 60% less message receptions than Deluge.

The simulation results in Figure 2.19-2.22 show that MNP outperforms Deluge in both completion time and energy consumption. We show that MNP performs well in both dense and sparse networks. For example, it takes a little more than 10 minutes to transmit a 5.6KB (2 segments) program to all the sensor nodes in a 20x20 network (either dense or sparse). And, a node's active radio time is only 20-43% of the completion time. By contrast, the performance of Deluge drops significantly when the network density increases, as shown in our simulations, as well as in [23] and [39]. This is caused by the discrepancy between the *observed* density by the nodes and the *physical* density (defined by the radio range), as pointed out in [39]. In Deluge (Trickle), nodes listen to the channel, and adjust their advertising frequency based on the *observed* density. When the network density increases, packets lose is more likely due to hidden terminal problem. This causes nodes to hear less traffic, and hence observe a lower density than the actual one (*physical* density). Unlike Deluge, the

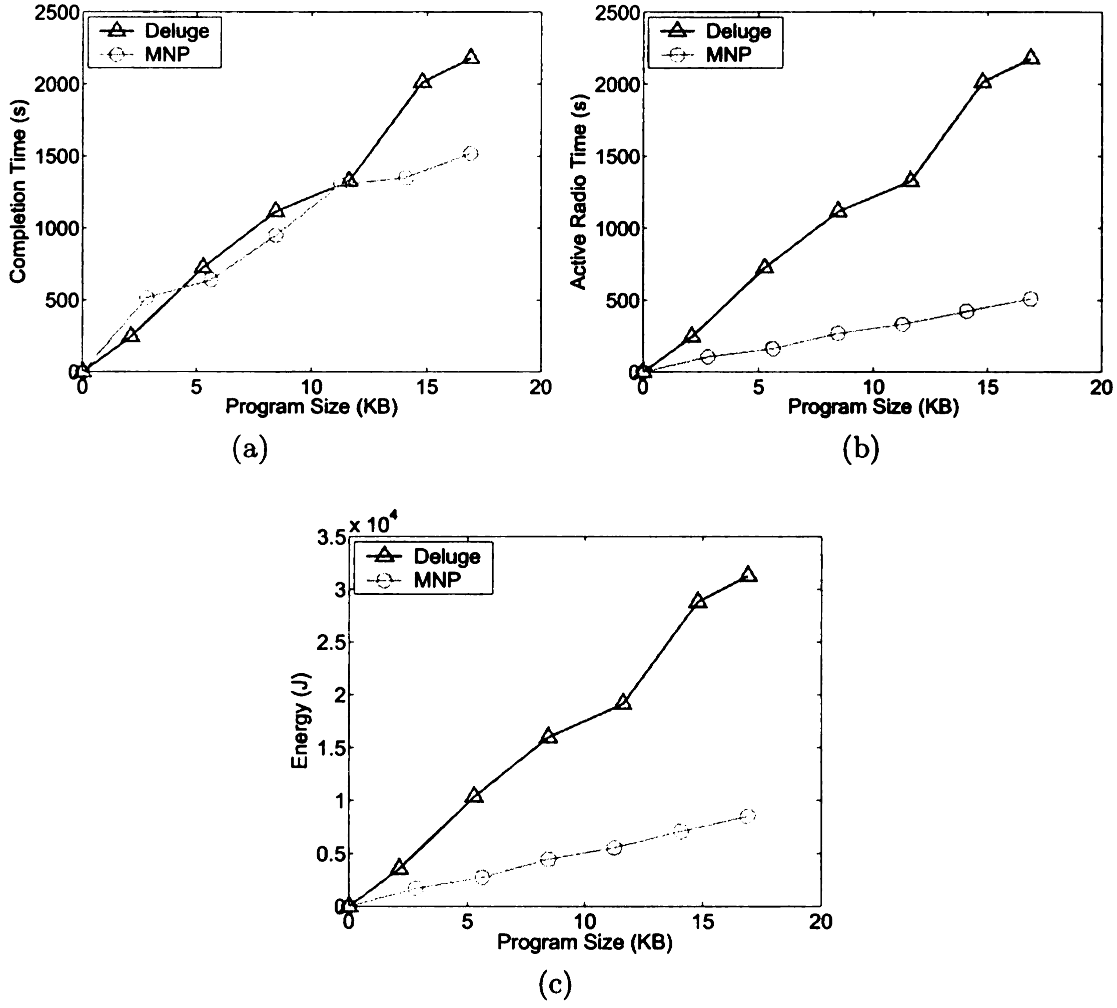


Figure 2.21: Performance in a sparse network. 20x20 network. Inter-node distance: 15 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

sender selection algorithm in MNP reduces the hidden terminal problem and message collision, hence, enables fast reprogramming.

Varying Network Sizes.

Based on observation from Figure 2.10, we use different contention sleep periods when the network size varies: 40 seconds for 10x10 and 12x12 networks, 80 seconds for 14x14 and 16x16 networks, and 160 seconds for 18x18 and 20x20 networks. In Figure 2.23, we show the completion time (a), the average active radio time per node (b), and the average energy consumption per node (c) when the network dimension

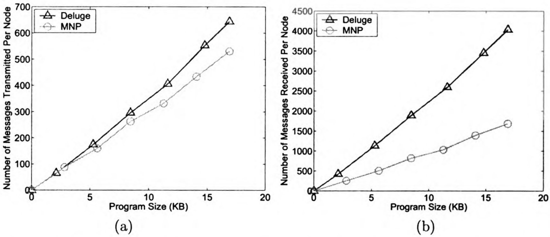


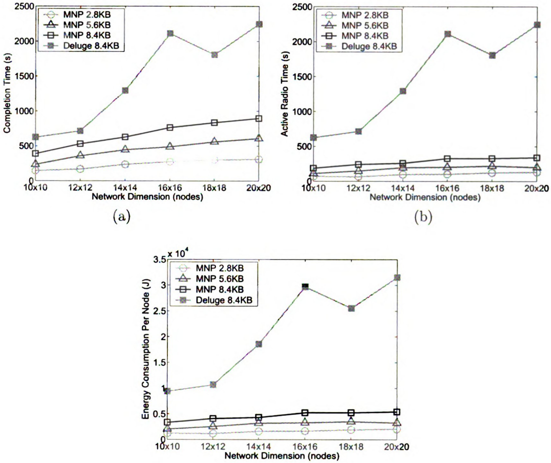
Figure 2.22: Transmissions and receptions in a sparse network. 20x20 network. Internode distance: 15 feet. (a) number of messages transmitted per node (b) number of messages received per node.

grows from 10x10 to 20x20. For comparison purpose, we also show the corresponding simulation results of Deluge in Figure 2.23.

As shown in Figure 2.23(a), the slope of completion time for transmitting multiple segments is close to that of a single segment, indicating the effectiveness of pipelining. We also notice that the average active radio time per node (b) and energy consumption per node increase gradually when the network size increases from 10x10 to 16x16, and stabilize at the same level when the network size is larger than 16x16. This indicates that a node is only affected by its local environment. After the network size has increased to a certain point, further increase does not increase the energy consumption. This shows that MNP is energy efficient and scales well to large network sizes. From Figure 2.23, we also note that to transmit a program of the same size, the time and energy required by MNP is much lower than that required by Deluge.

Placing Base Station(s).

Intuitively, by placing the base station in the center or adding more base stations, we expect that the completion time would be significantly reduced, because the base station(s) can serve as a starting point for each of the four 10x10 subnets. However,



(a)

(b)

(c)

Figure 2.23: Performance in a $N \times N$ network where $N = 10, 12, 14, 16, 18, 20$. Inter-node distance: 10 feet. Contention sleep period is 40s when $N = 10, 12$; contention sleep period is 80s when $N = 14, 16$; contention sleep period is 160s when $N = 18, 20$. (a) Completion time vs. network size (b) Average active radio time per node vs. network size (c) Average energy consumption per node vs. network size

it is not the case. As we have shown in Figure 2.12 , placing the base station in the center (or adding more base stations at the corners) only reduces the completion time by 12% (13%). Moreover, when putting the base station in the center (or adding more base stations at the corners), the active radio time and energy consumption even increase by 19% (14.3%) and 16.2% (12.3%) respectively.

The increase in active radio time is expected. Recall that the nodes that have the highest active radio time are those spending the most amount of time in *active updating* phase (cf. Section 2.2.2). When the base station is placed in the center or multiple base stations are placed at the corners, the distance from the base station to the farthest node is reduced by half, but the total reprogramming time is only affected a little. This means that nodes are activated earlier, but finish at about the same time; in other words, they spend more time in *active updating* phase. Therefore, the active radio time is longer.

We show the propagation progress for sending a two-segment program in a 20x20 network when the base station is placed in the center in Figure 2.24 (a)-(e). In Figure 2.24(f), we show the corresponding segment completion sequence in a 10-second window. Comparing with Figure 2.15 and Figure 2.16, we note that when the base station is placed in the center of the network, the overlapping area of Segment 1 and Segment 2 is smaller. This shows that when network diameter is reduced, pipelining effect is also reduced. There is a slow down in propagation when Segment 1 finishes and Segment 2 starts taking over. Recall one of the rules in pipelined version of MNP is that whenever a node is aware that one of its neighbors needs a lower segment or is going to transmit a lower segment, it will not advertise the next segment (cf. Section 2.1.1). When the base station is placed in the center, it has a larger neighbor set, thus is more severely suppressed. The base station starts transmitting Segment 2 only when it observes that all the neighbors have received Segment 1 and are ready to receive the next segment. Therefore, when data propagate from the center to the edges, although it takes about 40% less time to propagate each single segment, the reduction to the total completion time is not much due to the latency *between* segments.

In Figure 2.25, we show the propagation progress and segment completion sequence for sending a two-segment program in a 20x20 network when we put four base stations at the four corners of the network. Code segments propagate fast as they start from the corners, and slow down when they approach the center. This is because when the source nodes are far away from each other, they transmit data in separate communication cells simultaneously; as the propagation waves move to the center, the sources nodes' transmissions are suppressed by the sender selection algorithm, in order to avoid collision.

The previous simulations are done on a dense network with 10 feet node separation. We have also done simulations on a sparse network with 15 feet node separation, and the corresponding simulations for Deluge. We set the program size to 256 packets (5.63KB) for MNP and 240 packets (5.28KB) for Deluge. In Tables 2.3 and 2.4, we compare the performance of MNP and Deluge in a dense network and in a sparse network with different placements of the base station(s). We found that placing the base station in the center or adding more base stations at the corners reduces the completion time. And, the reduction is more obvious in a sparse network than in a dense network. For example, for MNP, moving the base station from the corner to the center reduces completion time by 12% in a dense network, and reduces completion time by 22% in a sparse network. For Deluge, moving the base station from the corner to the center reduces completion time by only 8% in a dense network, but reduces completion time by 37% in a sparse network. In Tables 2.3 and 2.4, we also show that MNP has lower energy consumption than Deluge, and it reprograms faster than Deluge does in a dense network.

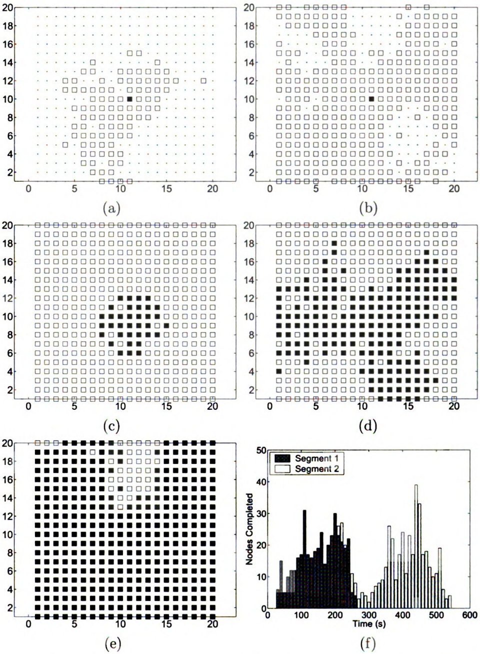


Figure 2.24: Propagation progress for sending a two-segment program in a 20x20 dense network. One base station is in the center. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). Contention sleep period: 80s. Total completion time: 536s. (a) 100s (b) 200s (c) 300s (d) 400s (e) 500s (f) segment completion sequence in a 10-second window. In (a)-(e), a node covered by an unfilled square means that it has received Segment 1; a node covered by a filled square means that it has received Segment 2.

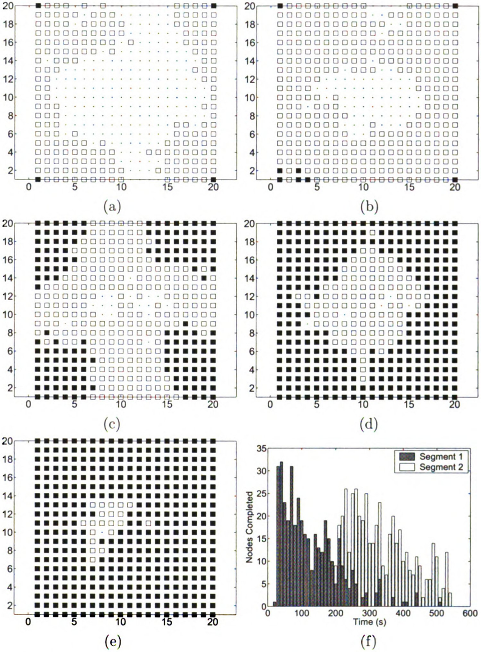


Figure 2.25: Propagation progress for sending a two-segment program in a 20x20 dense network. Four base stations are placed at the four corners. Inter-node distance: 10 feet. Program size: 2 segments (256 packets). Contention sleep period: 160s. Total completion time: 530s. (a) 100s (b) 200s (c) 300s (d) 400s (e) 500s (f) segment completion sequence in a 10-second window. In (a)-(e), a node covered by an unfilled square means that it has received Segment 1; a node covered by a filled square means that it has received Segment 2.

Table 2.3: The performance of MNP and Deluge in a dense network (inter-node distance is 10 feet). 20x20 network. Program size: 256 packets (5.63KB) for MNP, 240 packets (5.28KB) for Deluge. For MNP, contention sleep period is 160 seconds when the base station is placed at the corner, and 80 seconds when the base station is in the center or 4 base stations are places at four corners.

	MNP			Deluge		
	BS at one corner	BS in center	BSs at 4 corners	BS at one corner	BS in center	BSs at 4 corners
Completion time (s)	609	536	530	1114	1022	1183
Active radio time /node (s)	202	240	231	1114	1022	1183
Energy consumption /node (J)	3274	3804	3675	15732	14403	16757
Messages sent /node	84	99	109	181	124	273
Messages received /node	711	810	761	2463	1680	3300

Table 2.4: The performance of MNP and Deluge in a sparse network (inter-node distance is 15 feet). 20x20 network. Program size: 256 packets (5.63KB) for MNP, 240 packets (5.28KB) for Deluge. For MNP, contention sleep period is 40 seconds.

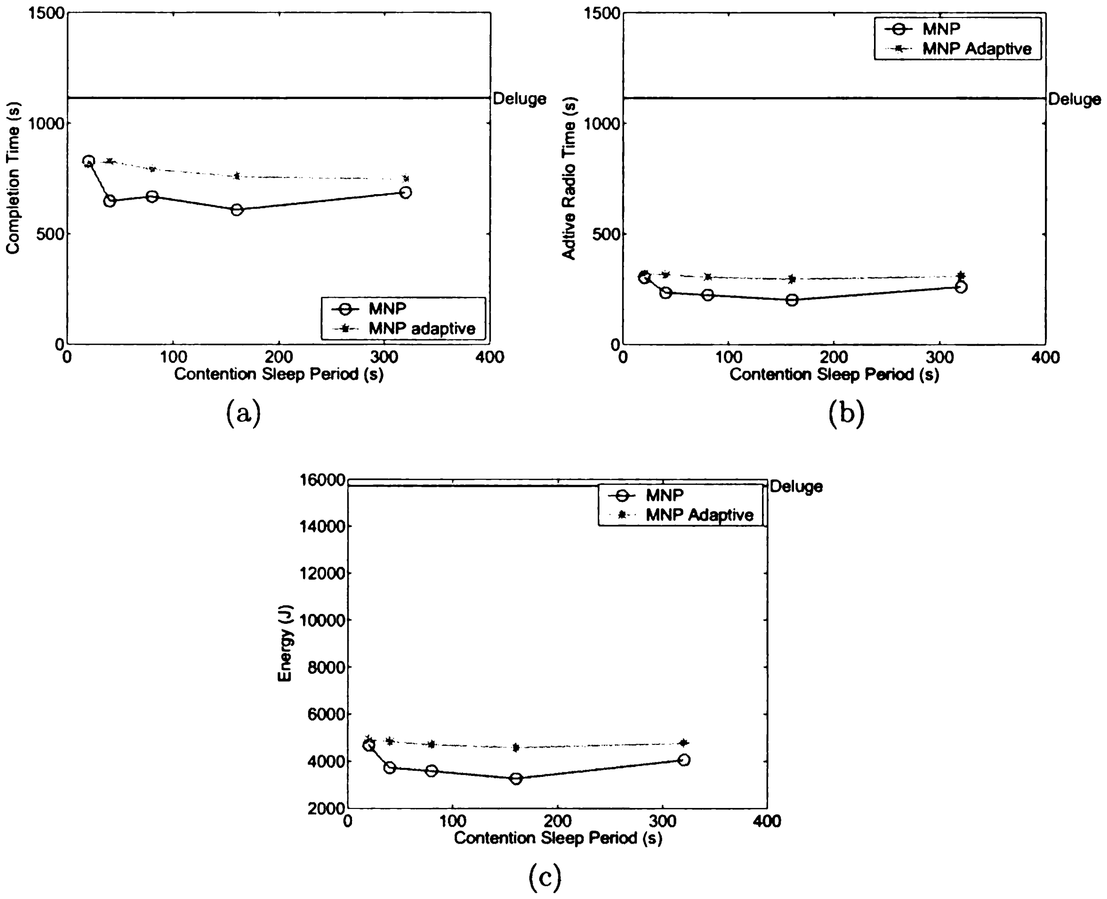
	MNP			Deluge		
	BS at one corner	BS in center	BSs at 4 corners	BS at one corner	BS in center	BSs at 4 corners
Completion time (s)	641	500	484	725	460	689
Active radio time /node (s)	163	164	157	725	460	689
Energy consumption /node (J)	2746	2761	2668	10365	6780	9899
Messages sent /node	160	155	170	174	173	210
Messages received /node	507	523	516	1132	1075	1259

Discussion: Contention Sleep Period

Our simulation results show that the performance of MNP varies when different contention sleep periods are used. It suggests that we should manually configure the contention sleep period based on the network density. It raises the question whether an automatically adjusted contention sleep period should be used instead of a manually determined one. To study this problem, we made a minor change to MNP. A node in contention sleep wakes up periodically (every 8 seconds), listens to the channel for a short amount of time (1 second), if, during this interval, it hears an advertisement (or a request) that contains the same segment ID (or higher) as the highest one it can provide, it goes back to sleep. Otherwise, it starts advertising (or sends a request), and hence, contention sleep ends. In this way, nodes adaptively adjust their contention sleep period based on their observed environment, which is similar to the approaches used in Deluge (Trickle).

We simulated the modified algorithm in 20x20 networks with 10 feet inter-node distance (dense networks) and with 15 feet inter-node distance (sparse networks). The program size is 5.6KB (256 packets). We show the completion time, average active radio time, average energy consumption of the modified algorithm (marked as MNP adaptive) under different contention sleep periods in Figure 2.26 (for the dense network case) and Figure 2.27 (for the sparse network case). For comparison, we also draw corresponding performance of MNP and Deluge on the two Figures. It shows that, with this modification, the reprogramming time and energy consumption are not affected by the assigned contention sleep period (since the period can be dynamically adjusted). In the dense network case (Figure 2.26), the modified algorithm (marked as MNP adaptive on the Figure) has higher completion time, active radio time and energy consumption than MNP when the contention sleep period is between 20 and 320 seconds, due to the increased message transmissions. In the sparse network case (Figure 2.27), the modified algorithm performances similarly as MNP when the contention sleep period is from 20 to 160 seconds.

If the network topology is unknown, we can always choose some default value for contention sleep period. Since contention sleep period only specifies the degree that transmissions should be suppressed, we expect that any value within 40 to 80 seconds range works reasonably well in all the network settings we have studied. In many cases we do have some knowledge about the network topology and radio range, we can utilize this knowledge to choose a more appropriate contention sleep period to improve the performance of MNP, e.g., increasing the contention sleep period when the network density is high.



(a)

(b)

(c)

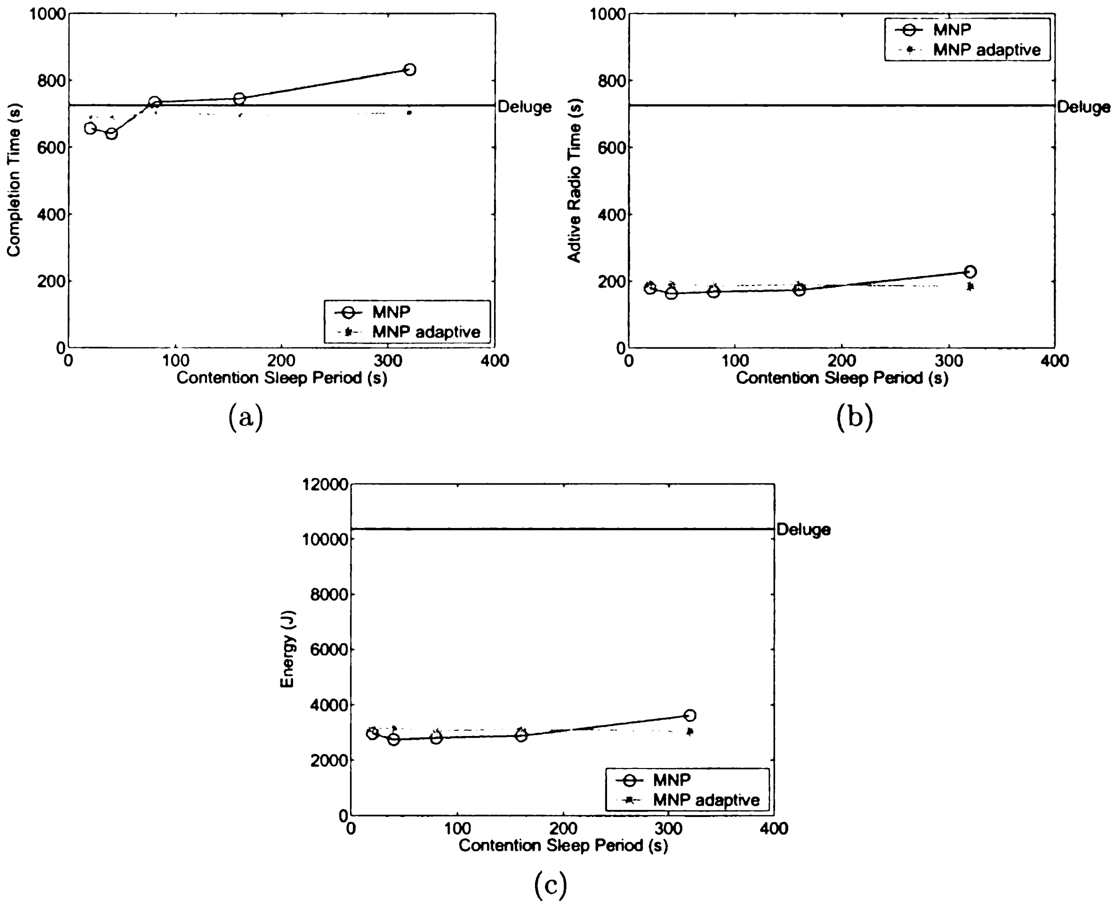
Figure 2.26: 20x20 network. Inter-node distance: 10 feet. Program size: 256 packets for MNP and MNP adaptive, 240 packets for Deluge. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

2.3 Discussion

In this section, we list the questions regarding MNP and try to provide answers.

Question: Since all nodes that receive an advertisement send a request (if they need the segment that is advertised), can it lead to significant collisions?

Answer: Before a node sends a request, it delays for a short random interval. This reduces message collisions. If a lot of sensors respond to a node's advertisements, the node does not need to record all of them (it can record up to 16 requesters, which is normally enough). In the worst case, the sender that is elected by the sender selection algorithm might not be optimal. But we do not guarantee it to be. Our



(a)

(b)

(c)

Figure 2.27: 20x20 network. Inter-node distance: 15 feet. Program size: 256 packets for MNP and MNP adaptive, 240 packets for Deluge. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

simulation results show that it works well in practice.

Question: MNP provides several parameters that can be tuned. How is performance affected if these parameters are chosen suboptimally?

Answer: Our simulation results show that even if the tunable parameters are chosen suboptimally, the performance of MNP is still reasonably well. For example, in a 20x20 dense network, the optimal contention sleep period is 160 seconds, but choosing any value in the range of 40 seconds to 320 seconds only affect the performance by maximally 12.9% (respectively, maximally 24.4%) in completion time (respectively, energy consumption).

When the network is unknown, we can use default values for parameters, or dynamically adapt parameters based on the observations from the network. Although the parameter values chosen are probably suboptimal, we can still preserve a large portion of energy savings. When network characteristics are known in advance, the designer can utilize that information to minimize energy usage (respectively, completion time) by choosing appropriate parameter values.

Question: Does MNP require full control of the radio during reprogramming?

Answer: While our simulations assume that there is no other traffic during reprogramming, it is possible that other traffic may exist. In the context of reprogramming, however, in most situations, the old application is to be replaced by a new one. Hence, it is reasonable to assume that the previous application is suspended. As long as the traffic from previous program is low, we expect that performance of MNP would not be significantly affected. In particular, such traffic may cause some additional packets of new program to be lost due to collision. However, if the rate of messages sent by previous application were low, it would not affect performance of MNP significantly.

Question: Does MNP depend on a specific MAC layer?

Answer: No. We have used the standard CSMA MAC layer provided by TinyOS platform. If the MAC layer that provides TDMA implementation (or 802.15.4), it would in fact improve the effectiveness of sender selection, as the MAC layer would improve the probability that a message gets to its destination. Hence, with such MAC layers, MNP will continue to work.

2.4 Chapter Summary

In this chapter, we presented a multihop network reprogramming protocol, MNP, that is targeted at Mica2/XSM motes. MNP uses a sender selection algorithm to reduce message collision and the hidden terminal problem. When multiple sensor nodes compete to become the sender, the sender selection algorithm attempts to find a node whose transmission of the program code is likely to have the most impact, and it tries to ensure that at a time at most one sender is active in any neighborhood. Also, MNP propagates the code in a pipelined fashion.

To improve energy efficiency of MNP, we identify the conditions that nodes can be put to sleep. In MNP, some nodes are selected to transmit the code whereas others can "sleep" to save power and to prevent interference. We call it *contention* sleep. Moreover, a node is put to sleep if none of its neighbors requests for the segment it is advertising. This type of sleep is called *noreq* sleep, and is used to reduce the idle listening during maintenance phase. We also introduce an optional *init* sleep to reduce the initial idle listening at the beginning of reprogramming. Through simulation on TOSSIM, we show the effect of varying contention sleep period, and the improvement on energy efficiency by adding noreq sleep and init sleep to MNP.

Furthermore, we studied the choice for the appropriate contention sleep period so that it effectively reduces message collision, without introducing too much latency. We conclude that the sleep period should be chosen according to network density. Our simulation showed that MNP performs well at any program sizes, network sizes, and network densities.

Moreover, we can adjust the power level used in the advertisement message based on the remaining battery level. Thus, a node whose battery level is low (e.g., if it became a sender in previous reprogramming) advertises with lower power level. Therefore, it is likely to have only a small number of requesters and, hence, it will lose in the sender selection. It follows that with this modification, the probability that a sensor node forwards the code to others depends on its remaining battery level. Therefore, the responsibility of transmitting the code will be evenly divided among the sensors.

MNP was demonstrated in the DARPA NEST team meeting in Columbus, OH, May 2004 and during the ExScal project demonstration in Avon Park, FL, December 2004 [61]. In the first demonstration, we deployed 50 Mica2 sensors on a grass field and reprogrammed all the sensors with Lites code [1]. In the second demonstration, we reprogrammed a network of 100 XSM sensors with a program of 22.4KB in 7-8 minutes.

Although MNP was designed as a code dissemination protocol, it can be used to disseminate any sort of data. By dividing the data into small segments, we allow incremental data updates. Moreover, in the scenario that several subsets of the network exist, rather than sending the data to the entire network, we can send different types of data to several disjoint or non-disjoint subsets of the network. In this case, our sender selection algorithm needs to be extended to take into account all these messages types, for example, giving different priorities to different types of messages.

Chapter 3

Gappa: Gossip Based Multi-channel Reprogramming for Sensor Networks

MNP (as well as all other existing reprogramming protocols, such as [23,30,47,60]) is designed for the scenario that one (or a few) sensor, which has the entire new program, is dropped on the field. This sensor then communicates the new program to the remaining sensors in the network. Another way suggested for reprogramming is with the help of an UAV (Unmanned Ariel Vehicle). Specifically, in this approach, an UAV flies over the network and communicates the new code to the sensors.

The UAV divides a program into multiple segments, and transmits the segments of code to the sensors when it flies over the network. Clearly, it is desirable that the contact time required for the UAV is short. Hence, it is likely that the sensors only get a part of the code from the UAV. One possible scenario is that at any time, only the sensors that are directly below the UAV receive the segment transmitted by the UAV at the time. Another possible scenario is that the UAV is equipped with a radio device that can communicate at multiple frequencies at once. In this case, the UAV can transmit each segment on a different frequency. The sensors themselves choose one of these frequencies and receive the corresponding segment.

In both scenarios mentioned above, each sensor (or a subset of sensors) is associated with one of the segments from the new program. The sensors then need to communicate the remaining segments with each other utilizing multiple radio channels. We denote this problem as the gossip based multi-channel reprogramming of sensor networks. In this approach, sensors are able to split the network traffic among different channels by exploiting multi-channel resources. Moreover, there is data redundancy, since every segment is associated with many sensor nodes. Compared to the type of communication that originates from one or a few seed nodes, this gossip based communication has the potential to enable higher concurrency and better utilization of channel capacity.

In this chapter, we propose *Gappa*, a gossip based multi-channel reprogramming protocol, for this gossip based reprogramming. *Gappa* utilizes multiple channels to rapidly and reliably reprogram all the sensors in the network. Based on channel allocation policies, we have two variations of *Gappa*, fixed-channel *Gappa* (*fc-Gappa*) and variable-channel *Gappa* (*vc-Gappa*). *fc-Gappa* uses a simple fixed channel allocation scheme, i.e., each segment is assigned one channel. On the other hand, in *vc-Gappa*, each sensor randomly selects a channel from all the available channels. The variable channel allocation scheme in *vc-Gappa* is a little more complicated than the fixed channel allocation, however, it allows better utilization of the radio channels.

The features of *Gappa* are as follows.

1. *Gappa* uses a multi-channel sender selection algorithm, which tries to guarantee that on each channel, only one sender is selected to transmit in a neighborhood at a time. Moreover, the algorithm attempts to select the sender whose transmission is expected to have the most impact on each channel. To better utilize multi-channel resources, if a node loses in the sender selection on one channel,

it will compete to transmit code on a different channel that is available. In this way, *Gappa* propagates code rapidly.

2. *Gappa* conserves energy by putting a sensor node to sleep if all the channels that it attempts to transmit code on are busy, and it is not interested in receiving the code segments that its neighbors are transmitting.

3. To enable gossip based communication, *Gappa* allows sensor nodes to receive segments that are out of order. There is less dependency on special nodes since every node that has a segment is a potential sender. This, combined with multi-channel usage and pipelining technique, leads to high concurrency in data exchange.

4. We propose two variations of *Gappa* that use different channel allocation policies, i.e., fixed channel allocation and variable channel allocation. We show that the variable channel allocation policy works better than the simple fixed channel allocation policy due to the fact that the sensors can utilize all the available channels.

We implement *Gappa* (the basic version and its extension) in TinyOS [22, 38] platform, and evaluate its performance using TOSSIM [37]. Through simulation, we compare with MNP and Deluge [23], and show that *Gappa* reduces the reprogramming time and energy consumption significantly. We also show that *vc-Gappa* performs better than *fc-Gappa* due to better channel utilization.

In Section 3.1, we present the gossip based multi-channel reprogramming protocol *Gappa*. We consider *fc-Gappa*, which uses a simple fixed channel allocation policy, in this section. In Section 3.2, we propose *vc-Gappa* that uses variable channel allocation. In Section 3.3, we evaluate the performance of fixed-channel *Gappa* (*fc-Gappa*) under different network settings. We also present the performance comparison with MNP and Deluge. In Section 3.4, we present the performance enhancement using variable channel allocation. We summarize the chapter in Section 3.5.

3.1 Protocol Description

In this section, we present *Gappa*. We focus on the variation of *Gappa* that uses fixed channel allocation (called fixed-channel *Gappa*, or *fc-Gappa*). The sensor nodes are equipped with a single radio interface, thus can communicate on one channel at a time. But they can switch to different channels at run time. The new program image that is to be deployed is divided into n segments (n is normally a small number from 1 to 20). Each segment has a fixed number of packets. We assume [1] that the number of available non-overlapping channels is at least $n + 1$. We select $n + 1$ non-overlapping channels, which are indexed from 0 to n. Without loss of generality, we define channel 0 as the *control channel* and channels 1 to n as the *data channels.* The control channel is used for transmitting the control messages (e.g., advertisements, requests), while the data channels are used for the actual data transmissions. Each data channel corresponds to one segment, i.e., segment k ($1 \leq k \leq n$) is always transmitted on channel k. The control channel is also the default channel, i.e., sensor nodes stay on channel 0 unless they are transmitting or receiving data packets of a certain segment.

Before we describe the algorithm in detail, we illustrate it using an example in Figure 3.1. The numbers marked on the sensor nodes represent the segments they have received. The edges represent communication links. We note several things from observing this simple network.

First, the nodes that have overlapping communication ranges cannot transmit the same segment simultaneously as it will cause significant collision on the shared data channel. For example, nodes A and B should not transmit segment 3 at the

[1] Note that this condition holds for most sensor platforms that are popularly used. For example, Mica2/Mica2Dot motes operating in the 902-928 MHz frequency band have 54 channels on which they can transmit [9].

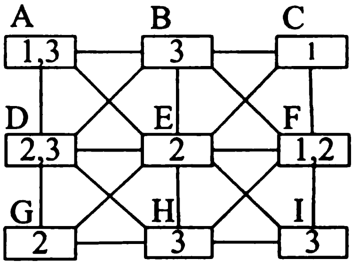


Figure 3.1: Example sensor network

same time.

Second, on each channel, the choice of the sender that transmits next is not uniform. If both nodes A and B want to transmit segment 3, B is a better choice than A, since more nodes in the neighborhood are expected to benefit from the transmission of node B.

Third, since sensor nodes can only communicate on one channel at a time, a node that has multiple segments must select one segment as the ***preferred segment***, and will transmit this segment if it is selected as the sender. The choice of the preferred segment is decided by the status of its neighbor nodes. For example, node D might choose segment 2 as the preferred segment, because more nodes in its neighborhood request for segment 2 rather than segment 3. However, if D finds that E has decided to transmit segment 2, D cannot transmit segment 2 simultaneously as it will cause collision with E on channel 2. In this case, transmitting segment 3 is a feasible alternative for node D.

Our algorithm consists of two parts: the control logic on the control channel and the operations on the data channels. We present these two parts in Section 3.1.1 and Section 3.1.2, respectively.

3.1.1 Operations on the Control Channel

Initially, all the sensor nodes are communicating on the control channel. Nodes perform two major tasks on the control channel: decide which nodes should switch to a data channel to perform data communication; and identify the nodes that are unlikely to contribute or receive data shortly and put them to sleep. The switching policy *tries to* guarantee that for each segment, at most one node in a neighborhood is selected to transmit the segment in the corresponding data channel. Moreover, it tries to select the sender that is expected to have the most impact. To achieve this goal, we extend the sender selection idea from MNP (cf. Chapter 2).

Multi-channel sender selection algorithm. In our algorithm, nodes perform sender selection using advertisements and requests. Each node maintains a sequence of <*SegID, ReqCtr*> pairs that indicate the segments it has received and the corresponding numbers of distinct requests (from different requesters) for those segments the node has received so far. *ReqCtrs* for all the segments are set to 0 when a node starts advertising. When a node receives a request that is *destined* to it from a "new" requester, it increments the *ReqCtrs* for the requested segments by one. Additionally, a node maintains a *preferred segment* ID, which is the segment that is requested by most number of nodes, i.e., has the highest *ReqCtr*. The preferred segment ID is set to 0 if the node has not received any request, and is recalculated whenever the *ReqCtrs* change. In the case that there are more than one segments have the highest *ReqCtr*, the preferred segment is randomly selected from these segments.

A node advertises the segments it has received, its preferred segment, as well as the *ReqCtrs* for all the received segments. Hence, an advertisement message includes the sequence of <*SegID, ReqCtr*> pairs, the preferred segment ID, and other information (program ID and size, source ID). When a node, say j, receives an advertisement message from a node, say k, if j needs any of the segments that are advertised, then it sends a request to k. The request message sent by j contains not only the IDs of the segments j expects to receive from k, but also k's preferred segment ID and the corresponding *ReqCtr* of k for that preferred segment (computed from the sequence of <*SegID, ReqCtr*> pairs that k sent in the advertisement message). While the request is intended for k, it is sent as a broadcast message with k as one of the fields. Thus, when another node, say l, receives the request, l is aware of the fact that k is a potential sender. This allows us to count for the hidden terminal effect where l could not have received the advertisement message from k.

We note that a node sends a request to all senders that send the advertisement messages containing the code segments the node is interested in. This ensures that a node is aware of all the requesters who are likely to receive the code if it is chosen to transmit the code. Moreover, the sender selection is performed only among nodes that have the same preferred segment. (For example, if a node, say k, has preferred segment 3, and its neighbor l has preferred segment 1, they can transmit simultaneously on different channels (channel 3 and 1) without interrupting each other.) If k loses to l that has more requesters for the preferred segment, k takes the two actions in Figure 3.2.

1. Mask the preferred segment, and check to see if it has any segments that are not masked. If so, start advertising the remaining segments (and try to transmit on a different channel that is currently available). Otherwise, turn to idle stage.
2. Reset the *ReqCtrs* of all its segments to 0. Reset its preferred segment ID to 0. (This is due to the fact that some of the old requesters of this node may be receiving code from the node that wins the sender selection on a different channel.)

Figure 3.2: Actions taken by a node that loses in the sender selection.

A node on the control channel is in one of the two stages: advertise stage and idle stage. Initially, if a node has a segment that it can advertise, then it is in advertise stage; otherwise, it is in idle stage.

Tasks in advertise stage. A node S in advertise stage broadcasts an advertisement message every random interval (we use random interval to avoid message collision). It reacts to the requests, advertisements, and "SwitchChannel" messages as described next.

1. *Actions taken by an advertising node on receiving a request.* Every time S receives a request message, it checks to see if this message is *destined* to it. If the message is destined to it, and is from a "new" requester that S has not seen before, S increments the *ReqCtrs* of the requested segments by one, and recalculates its preferred segment ID. If the request message is destined to another node Q, Q has the same preferred segment ID as S and Q has more requesters, then S loses in the sender selection, and will take the actions in Figure 3.2.

2. *Actions taken by an advertising node on receiving an advertisement.* When S receives an advertisement message from a node Q, if S needs any of the segments Q advertises, S broadcasts a request message destined to Q after a short random interval (to avoid collision with other request messages sent to Q). As mentioned earlier, it puts the ID and *ReqCtr* of Q's preferred segment in the request message. In addition, S also checks to see if it loses to Q in the sender selection algorithm. If so, S takes the actions in Figure 3.2. Note that this sender selection procedure cannot cause deadlock, as the node with the highest *ReqCtr* of the preferred segment - with appropriate tie breaker on node ID - will succeed.

3. *Actions taken by an advertising node on receiving a "SwitchChannel" message.* If S receives a "SwitchChannel" message from a node Q, which indicates that Q is going to transmit its preferred segment in the corresponding data channel. If S is interested in receiving the segment from Q, then S switches to the data

channel for that segment. Otherwise, if the segment that Q is going to transmit is the same as S's preferred segment, which means that S has lost the sender selection for this segment, then S takes the actions in Figure 3.2.

The advertise stage ends when a node has sent a given number of advertisements continuously (without resetting *ReqCtrs* or switching channels). At this point, if it has received one or more requests, it broadcasts "SwitchChannel" messages and switches to the data channel that is assigned to its preferred segment. Otherwise, it turns to idle stage.

Tasks in idle stage. A node in idle stage can choose to keep its radio on to listen to the channel, or turn its radio off to save energy. Thus, a node in idle stage is in one of the two states: listen state (with radio on) and sleep state (with radio off). The length of time t a node stays in idle stage, and whether it keeps its radio on (listen or sleep) when it is idle, are decided by the status of the node and its observation of neighbors. Specifically, a node maintains two boolean variables: *TendToReceive* and *TendToSend*, which indicates the node's intention to receive or transmit a segment. *TendToReceive* is set to *false* initially and when a node has successfully received a complete segment. When the node hears an advertisement, request, or "SwitchChannel" message, it checks to see if it needs any of the segments that are advertised (or requested) in the message. If so, the node sets its *TendToReceive* to *true*. *TendToSend* is set to *false* when a node starts advertising. When the node hears an advertisement or request message, if it finds that it has some segments that other nodes do not have (requested or not advertised), it sets its *TendToSend* to *true*.

When a node enters idle stage, if it has received the entire program, or its *TendToReceive* is *false*, then it goes to sleep state. Otherwise, it is in listen state. A node in sleep state does not sleep through the entire idle stage. Rather, it takes short naps (say, 4s), wakes up and checks the channel for a short amount of time (say, 0.5s) between naps. If messages received during this interval causes *TendToReceive* to become *true*, the node turns to listen state, and keeps its radio on for the rest of time (in idle stage).

The length of idle stage t is exponentially increased, starting with a minimum t_l to a maximum t_u. This allows us to dynamically adjust the rate of advertising: nodes advertise aggressively when reprogramming is actively in progress and advertise slowly to save energy when most nodes have received the code. t is reset to t_l in two situations. First, if a node switches to a data channel (to transmit or receive data), when it returns to the control channel, it will reset t to t_l. Second, if a node receives a request, or its *TendToSend* becomes *true* (i.e., it identifies *potential requesters*), it sets t to t_l.

Although a node in idle stage does not advertise or participate in sender selection, it still sends requests or switches channel when needed. When a node S is in listen state, it reacts to the advertisements and "SwitchChannel" messages as described next.

1. *Actions taken by a node in listen state on receiving an advertisement.* If S receives an advertisement message that advertises segments it is interested in, S broadcasts a request message destined to that advertising node.

2. *Actions taken by a node in listen state on receiving a "SwitchChannel" message.* If S receives a "SwitchChannel" message that contains the ID of the segment S is interested in receiving, S switches to the data channel for that segment, and as a result, the idle stage ends.

3.1.2 Operations on Data Channels

When a node S decides to become a sender, it broadcasts a "SwitchChannel" message for a few times, then switches to the data channel that is assigned to its preferred segment. When a neighbor node hears a "SwitchChannel" message from S, if it needs the segment, it will switch to the corresponding data channel, and turn to *download* state.

We note that although the sender selection algorithm attempts to keep only one active sender in a given neighborhood on each channel, it is possible to have multiple active senders due to time-varying link properties. Hence, when S enters the data channel, it listens to the radio for a short amount of time (say, 1s), which we call *pre-forward* state, before it starts forwarding data. If S hears any message when it is in pre-forward state, it realizes that another node is currently transmitting data on that data channel. Hence, it will mask the segment it is going to transmit (to ensure that it will not reenter this channel immediately), return to the control channel, and start advertising the remaining segments. In this case, those nodes that have followed S to this data channel (switched to this data channel after receiving S's "SwitchChannel" messages) are left on this channel. If the nodes are able to receive packets from the current sender, they will stay on this channel and receive data from the current sender. Otherwise, they will return to the control channel after a time out.

Gappa uses the a similar loss recovery mechanism as MNP. Each packet has a unique ID. Each node maintains a bitmap, which we call *MissingVector*, for each of the segments it is receiving. Each bit in a *MissingVector* corresponds to a packet. All bits are initially set to 1. When a node receives a packet in a segment for the first time, it stores that packet in EEPROM (external storage for motes), and sets the corresponding bit in the *MissingVector* for that segment to 0. In this way, we guarantee that each packet in a segment is written to EEPROM only once. Note that *Gappa* allows nodes to receive segments that are out of order. Thus, a node might have received several incomplete segments. It is necessary for a node to maintain bitmaps (*MissingVectors*) for all the segments it has not completely received. For simplicity, we assume that the *MissingVectors* are in memory. We note that the extension for storing them on EEPROM and loading only the *MissingVector* for the segment that is being received in memory is straightforward.

Each node also maintains a *ForwardVector*, which is a bitmap of the segment that it is going to transmit, and is an indicator of the packets the node needs to send. When the pre-forward stage times out, the sender node S turns to *forward* state, and broadcasts a "StartDownload" message several times. S includes its *ForwardVector*, which is initially set to 0 (i.e., all the bits are set to 0), in the "StartDownload" message. When a node hears a "StartDownload" message, it waits for a short random interval (to avoid collision with transmissions from other requesters), checks to see if the *ForwardVector* contained in the "StartDownload" message has already included all the packets it needs. If so, it keeps silent. Otherwise, it sends a "RequestPackets" message to S. The "RequestPackets" message contains its loss information (*MissingVector*) for this segment. When S receives a "RequestPackets" message, it unions its *ForwardVector* with the *MissingVector* contained in the message. This updated *ForwardVector* is included in the "StartDownload" message that S sends next time. In this way, S's neighbor nodes are aware of the packets S is going to send, and hence, will not send requests repeatedly. We restrict the length of the segment to be no longer than 128 packets, so that the maximal size of *MissingVector* and *ForwardVector* is only 16 bytes, and thus fits into a radio packet.

After S has transmitted the "StartDownload" message for a few times, it starts transmitting the packets indicated in its *ForwardVector*. The download process ends when the receiver receives an "EndDownload" message from the sender. At this point, if the node has successfully received the whole segment, it includes the segment in the sequence it will use in future advertisements. When the download process ends, both the sender and the receivers return to the control channel, and restart advertising.

It is possible that the receiver never gets the "EndDownload" message. The reason can be the sender dies or returns to the control channel during pre-forward stage, or the "EndDownload" messages collide with other messages. To avoid being stuck in *download* state, a node in *download* state always sets a timer when it is waiting for the next packet. If the timer expires, it returns to the control channel.

As we mentioned earlier, a node masks a segment when it loses in the sender selection for that segment, or when it detects a busy data channel in *pre-forward* state. In both cases, the node cannot advertise or transmit this segment until the other node has finished transmitting the segment. Since this node does not know the exact time when the other node finishes transmitting, it keeps the segment masked for a certain amount of time. The mask bits are cleared when a node turns to idle stage or starts forwarding packets on a data channel.

3.1.3 The State Machine

In Figure 3.3, we show an overall picture of ***Gappa*** (***fc-Gappa***). ***Gappa*** operates as a state machine. Also, pseudo code of tasks in advertise stage on control channel can be found in Figure 3.4.

3.2 Variable Channel Allocation

The fixed-channel ***Gappa*** (***fc-Gappa***) assigns each segment a separate channel. Although this approach is simple and straightforward, sensors do not make full use of the available channels. Consider the scenario that two neighboring sensors intend to transmit the same segment. As the two sensors compete for the same radio channel, only one of them is selected to transmit data at a time, even though multiple channels are available. To make better use of the available channels, we make the following modifications to the algorithm in Sections 3.1.1 and 3.1.2.

1. A sensor *randomly* selects a data channel among all the available (not masked) data channels, and includes the channel number in its advertisement and

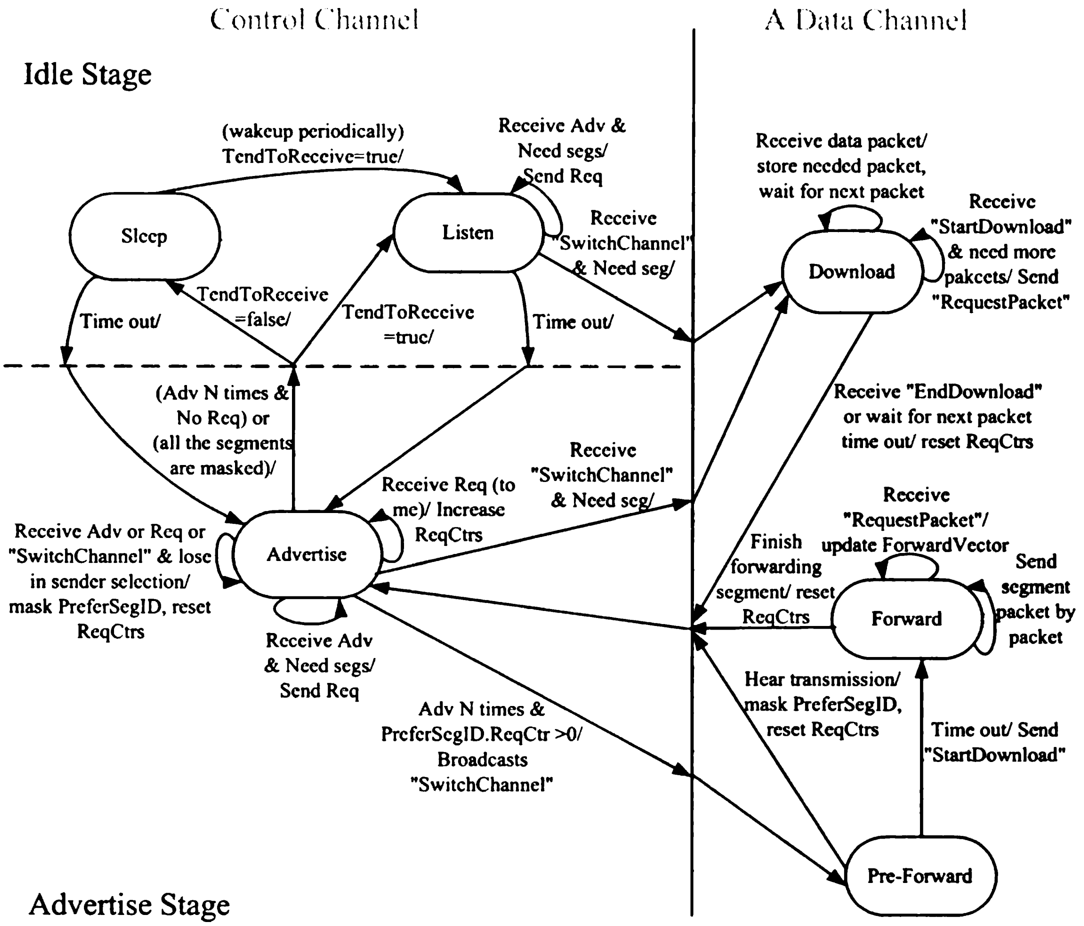

Control Channel
A Data Channel
Idle Stage
(wakeup periodically) TendToReceive=true/
Receive Adv & Need segs/ Send Req
Sleep
Listen
Receive "SwitchChannel" & Need seg/
Time out/
TendToReceive =false/
TendToReceive =true/
Time out/
Receive data packet/ store needed packet, wait for next packet
Download
Receive "StartDownload" & need more pakcets/ Send "RequestPacket"
(Adv N times & No Req) or (all the segments are masked)/
Receive Req (to me)/ Increase ReqCtrs
Receive "SwitchChannel" & Need seg/
Receive "EndDownload" or wait for next packet time out/ reset ReqCtrs
Receive Adv or Req or "SwitchChannel" & lose in sender selection/ mask PreferSegID, reset ReqCtrs
Advertise
Receive Adv & Need segs/ Send Req
Receive "RequestPacket"/ update ForwardVector
Finish forwarding segment/ reset ReqCtrs
Forward
Send segment packet by packet
Adv N times & PreferSegID.ReqCtr >0/ Broadcasts "SwitchChannel"
Hear transmission/ mask PreferSegID, reset ReqCtrs
Time out/ Send "StartDownload"
Pre-Forward
Advertise Stage

Figure 3.3: *Gappa* (*fc-Gappa*): the state machine.

"SwitchChannel" messages.

2. When a sensor sends a request to an advertising node, it includes the data channel number of the advertising node in the request message.

3. In Figure 3.2, the first action taken by a sensor that loses in the sender selection needs to modified as follows.

   - Mask the winner's data channel, stop for a while, and restart advertising. When a sensor restarts advertising, it checks to see if its data channel is masked. If so, it randomly selects another data channel that is available. If all the channels are occupied, it turns to idle stage.

Figure 3.4: Pseudo Code for Tasks in Advertise Stage on Control Channel

```
Broadcast an advertisement message every random interval
After advertising N times (without resetting ReqCtrs or switching channels):
    if (my.PreferSegID.ReqCtr > 0)
        Broadcast "SwitchChannel" messages, switch to Channel(my.PreferSegID).
    else
        Enter idle stage.
    endif

(a)
if a request message ReqMsg arrives
    if (ReqMsg.DestID == my.ID)
        if ( IsNew(ReqMsg.SourceID) )
            Increment ReqCtrs of the requested segments by 1, update my.PreferSegID.
        endif
    else //the message is destined to some other node
        if (ReqMsg.PreferSegID.ReqCtr > 0) and (ReqMsg.PreferSegID == my.PreferSegID) and
        ((ReqMsg.PreferSegID.ReqCtr > my.PreferSegID.ReqCtr ) or
        (ReqMsg.PreferSegID.ReqCtr == my.PreferSegID.ReqCtr) and (ReqMsg.DestID>my.ID))
            Mask my.PreferSegID, reset all ReqCtrs to 0, my.PreferSegID = 0
            if (there are segments not masked)
                Restart advertising the remaining segments.
            else
                Enter idle stage.
            endif
        endif
    endif
endif

(b)
if an advertisement message AdvMsg arrives
    (b1)
    if (Need(any of AdvMsg.SegIDs))
        Prepare request message ReqMsg
            ReqMsg.DestID = AdvMsg.SourceID
            ReqMsg.PreferSegID = AdvMsg.PreferSegID
            Find AdvMsg.PreferSegID.ReqCtr from the <SegID, ReqCtr> pairs contained in AdvMsg
            ReqMsg.PreferSegID.ReqCtr = AdvMsg.PreferSegID.ReqCtr
        Send ReqMsg after a short random time
    endif
    (b2)
    if (AdvMsg.PreferSegID.ReqCtr > 0) and (AdvMsg.PreferSegID == my.PreferSegID)
    ((AdvMsg.PreferSegID.ReqCtr > my.PreferSegID.ReqCtr) or
    (AdvMsg.PreferSegID.ReqCtr == my.PreferSegID.ReqCtr) and (AdvMsg.SourceID>my.ID))
        Mask my.PreferSegID, reset all ReqCtrs to 0, my.PreferSegID = 0
        if (there are segments not masked)
            Restart advertising the remaining segments.
        else
            Enter idle stage.
        endif
    endif
endif

(c)
if a "SwitchChannel" message SwitchMsg arrives
    if (Need(SwitchMsg.SegID))
        Switch to Channel(SwitchMsg.SegID)
    else if (SwitchMsg.SegID == my.PreferSegID)
        Mask my.PreferSegID, reset all ReqCtrs to 0, my.PreferSegID = 0
        if (there are segments not masked)
            Restart advertising the remaining segments.
        else
            Enter idle stage.
        endif
    endif
endif
```

4. On a data channel, if a sensor in *pre-forward* stage hears any message, which means that another sensor is transmitting code on that channel, it masks the channel, and returns to the control channel. Later when it starts advertising in the control channel, it randomly selects a different data channel that is not masked.

We Note that there are two major changes in variable-channel *Gappa* (or *vc-Gappa*). First, the data channel is randomly selected, and is not related to the segments a sensor has. This allows maximal utilization of the available channels. Second, when a sensor loses in the sender selection algorithm, instead of masking the segment, it masks the data channel. In other words, the sensor can transmit the same segment on a different channel. Hence, a sensor always advertises all the segments it has. Also note that if a sensor loses in the sender selection algorithm, it needs to pause for a short duration before it restarts advertising. This duration allows the winner of the sender selection, as well as its followers, to switch to its data channel without being interrupted.

3.3 Evaluation

We implement *Gappa* on TinyOS platform, and evaluate it using TOSSIM [37]. Radio transmission in TOSSIM is simulated as follows. Each node maintains a variable *radio_active*, which is set to 0 at the initial state. Every time a node transmits a bit, it increments the *radio_active* values of all its neighbors. In a lossy model, the transmitted bit can be flipped if a bit error occurs. When a node finishes transmitting, it decrements the *radio_active* values of all its neighbors. A node hears a bit if its *radio_active* value is 1 or greater. Although TOSSIM only models radio transmission on a shared channel, we can make a few changes so that it also simulates radio transmission on multiple channels. Towards this end, each sensor node maintains an additional *frequency* variable, which is the channel number the node is currently communicating on. When a node transmits or stops transmitting, it only modifies (increments or decrements) the *radio_active* values of those neighbors that are on the same frequency. Moreover, when a node switches to a different channel (i.e., changes its *frequency* variable), its *radio_active* variable is reset to 0. Note that the switching channel action can be taken only when the radio transmission of the node on the current channel is completed.

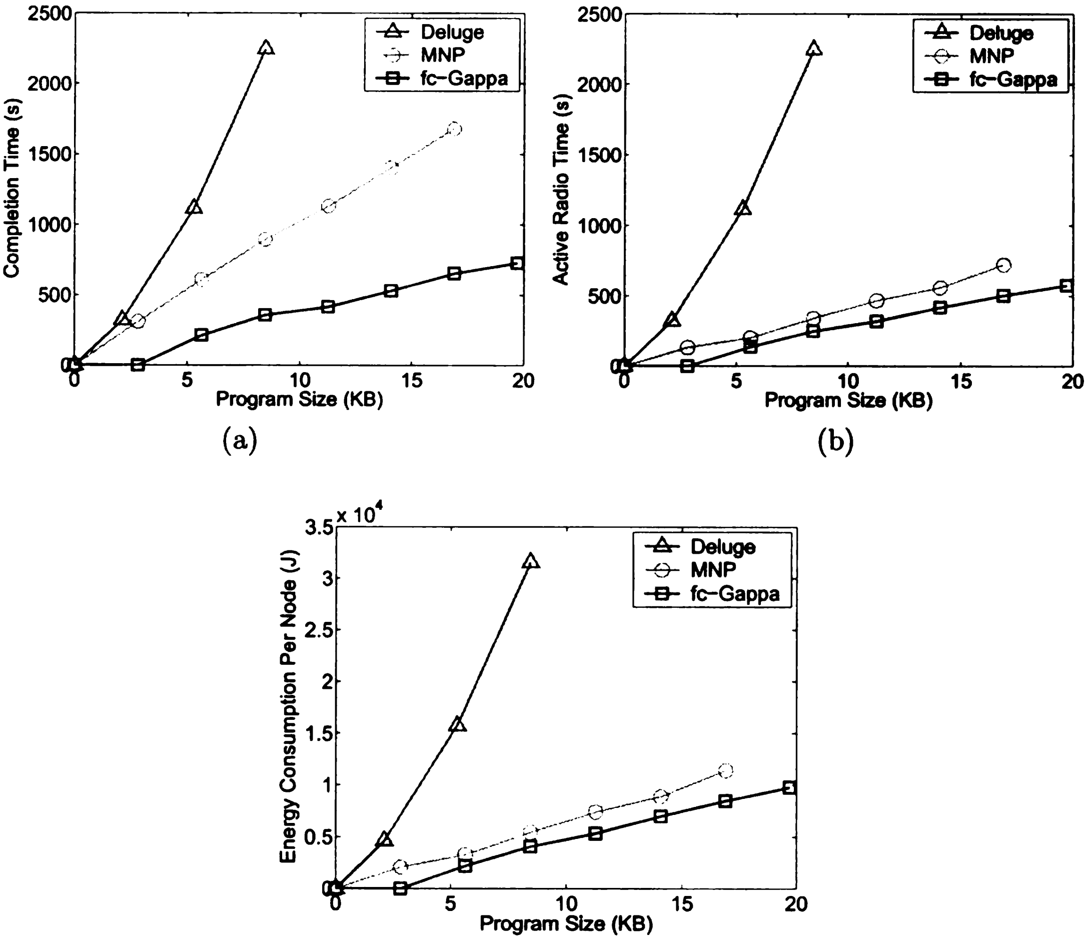
We calculate the energy consumption by counting the operations performed during reprogramming. (Alternatively, we can also use PowerTossim [58] to evaluate power consumption. However, since each simulation lasts for tens of hours, the trace file generated during simulation, which is required by PowerTossim in order to compute the energy usage, becomes too large (of the order of several gigabytes) to process.) We use Equation 2.1 from Section 2.2.2 in Chapter 2 to compute the energy consumption.

In this section, we evaluate fixed-channel *Gappa* (*fc-Gappa*), and compare it with MNP (cf. Chapter 2) and Deluge [23]. In the current implementation, each segment has 128 data packets. t_l and t_u are set to 16 seconds and 512 seconds respectively. The simulations are performed in a grid topology. Due to the fact that the execution time of each simulation is of order of tens of hours, we do not provide confidence intervals. In Sections 3.3.1 and 3.3.2, we show how the algorithm performs under various network settings, specifically program sizes, network densities, network sizes. In these sections, we assume that initially all the sensors have received one segment (which is randomly decided) from an UAV. In Section 3.3.3, we consider the situation where only a subset of the sensors initially have a segment.

3.3.1 Varying Program Sizes and Network Densities

Simulation setup 1. (Dense network) In the first set of simulations, we set the distance between every two neighbor nodes to 10 feet (a dense network). The simulations were performed in a 20x20 grid topology. We run *fc-Gappa*, MNP and Deluge under the same network settings. For MNP and Deluge, we assume that initially only the base station, the node at the bottom-left corner, has the new program. For *fc-Gappa*, every node has one segment (randomly chosen from 1 to n, suppose the program has n segments) initially. In Figure 3.5, we compare the completion time, average active radio time per node, and the average energy consumption per node of these three protocols, under different program sizes. We find that with the increase of program size, the completion time, average active radio time per node, and average energy consumption increase for all these protocols. Among the three protocols, *fc-Gappa* has the lowest completion time and energy consumption. To disseminate a program of the same size, *fc-Gappa* saves 60-65% completion time and 22-33% energy compared to MNP, and saves 81-84% completion time and 86-88% energy compared to Deluge. In Figure 3.6, we show the average number of transmissions and receptions per node in there three protocols. We notice that the number of transmissions of *fc-Gappa* is higher than the other two schemes. This is expected, as gossip based communication allows every node in the network to talk to each other, rather than require most nodes to listen to a few elected senders. Moreover, in *fc-Gappa*, multiple senders are transmitting code in different data channels at the same time, while a receiver can only receive code in one data channel at a time. As a result, redundancy increases. Although the overall traffic increases, the traffic is diverted to different channels.

Simulation setup 2. (Sparse network) We repeated the same set of simulations for *fc-Gappa*, MNP and Deluge on a sparse network, where the distance between two neighbor nodes is 15 feet. We show the completion time, the average active radio time per node, and the energy consumption per node of these three protocols in Figure 3.7. The results are similar. To reprogram the network with a program of the



(a)

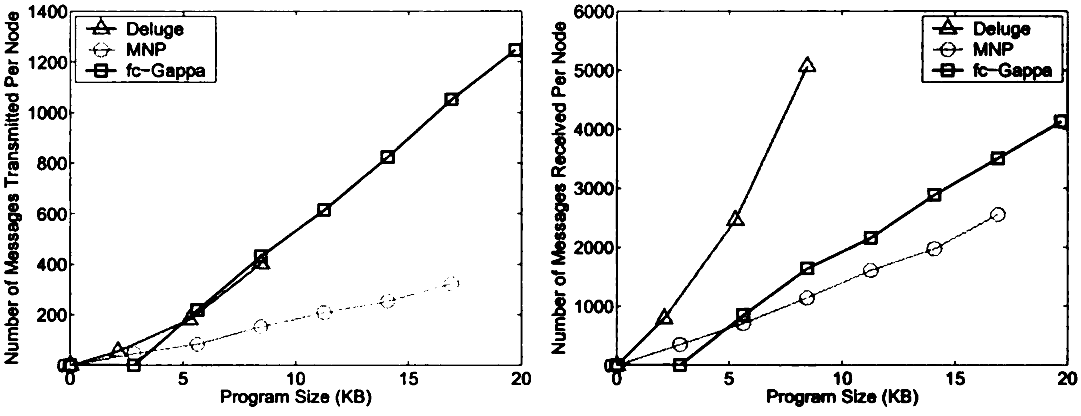
(b)

(c)

Figure 3.5: Inter-node distance: 10 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

same size, *fc-Gappa* saves 62-70% completion time and 17-26% energy compared to MNP, and saves 69-75% completion time and 73-81% energy compared to Deluge. In Figure 3.8, we show the average number of transmissions and receptions per node. *fc-Gappa* has higher transmissions than the other two protocols.

Simulation setup 3. (Base station in the center) Intuitively, the performance of MNP and Deluge will be better if the base station is placed in the center of the network. Therefore, we also simulated the case where the node that has the new program (base station) is in the center of the network. We set the program size to



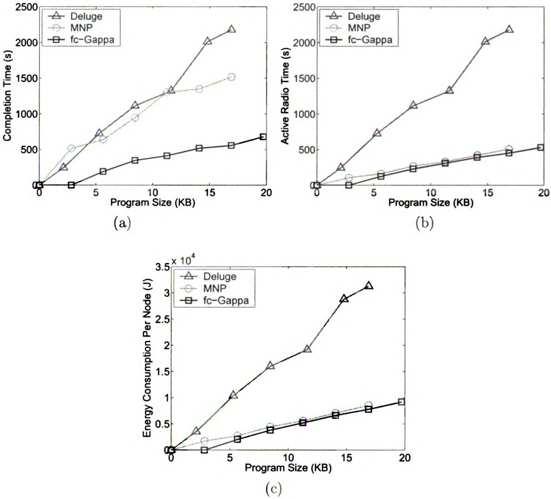
(a) (b)

Figure 3.6: Inter-node distance: 10 feet. (a) number of messages transmitted per node (b) number of messages received per node.

384 packets (8.45KB) for MNP and Deluge. For *fc-Gappa*, we set the program size to 512 packets (4 segments, 11.26KB). This way, in *fc-Gappa*, each node needs to receive 384 packets (3 segments). We present simulations results in Table 3.1 and 3.2. We can see that *fc-Gappa* has lower completion time even if the base station is in the center of the network (for MNP and Deluge).

Simulation setup 4. (Multiple base stations) We note that in a 20x20 network, the performance of MNP and Deluge does not improve significantly even if multiple base stations are used, due to the fact that excessive contention occurs when the propagation waves from different base stations meet. For example, for MNP, to transmit a program of 256 packets, the completion time when 4 base stations are placed at the corners (each placed at one corner) is 530 seconds, which is almost the same as the completion time when one base station is placed in the center (536 seconds). Similar results hold for Deluge. For this reason and due to limitation of space, we do not present the simulation data for the multiple base stations case.

Additional observations from simulation setups 1-2. In addition to the average values, we also study the distributions of active radio time and radio communication. We consider the case where the program size is 5 segments (14.08KB,



(a)

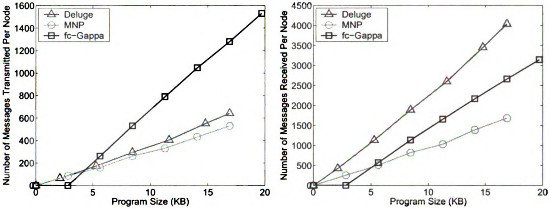
(b)

(c)

Figure 3.7: Inter-node distance: 15 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

Table 3.1: The base station node is in the center of the network or at the corner. The dense case: inter-node distance is 10 feet. Program size: 384 packets (8.45KB) for MNP and Deluge, 512 packets (11.26KB) for *fc-Gappa*.

	MNP center	MNP corner	Deluge center	Deluge corner	*fc-Gappa*
Completion time (s)	830	896	1585	2247	417
Active radio time (s)	390	341	1585	2247	320
Transmissions per node	152	153	216	402	614
Receptions per node	1221	1146	2898	5067	2162
Energy consumption per node (J)	6101	5429	22382	31551	5338



(a) (b)

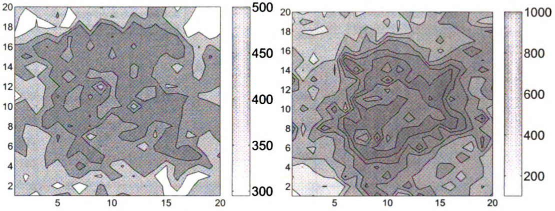
Figure 3.8: Inter-node distance: 15 feet. (a) number of messages transmitted per node (b) number of messages received per node.

Table 3.2: The base station node is in the center of the network or at the corner. The sparse case: inter-node distance is 15 feet. Program size: 384 packets (8.45KB) for MNP and Deluge, 512 packets (11.26KB) for *fc-Gappa*.

	MNP center	MNP corner	Deluge center	Deluge corner	*fc-Gappa*
Completion time (s)	744	949	905	1115	413
Active radio time (s)	269	270	905	1115	310
Transmissions per node	237	263	273	296	790
Receptions per node	803	822	1694	1891	1657
Energy consumption per node (J)	4445	4465	13122	15982	5202

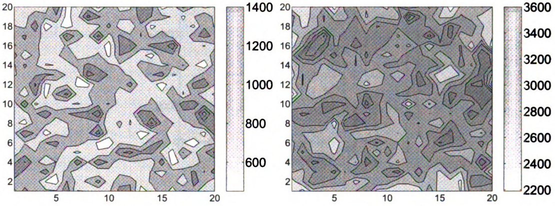
640 packets). In Figure 3.9, we compare the active radio time distribution of ***fc-Gappa*** and MNP (For Deluge, all the nodes's active radio time is the same as completion time). We note that the distribution of nodes' active radio time in ***fc-Gappa*** is more even (ranges from 300-500s) than the distribution of nodes' active radio time in MNP (ranges from 200-1000s).

In Figure 3.10(a), we show the distribution of transmissions. Some nodes transmit more than others. These nodes are *distinct* senders, i.e., they are selected as senders many times. Note that these distinct senders are randomly distributed. In Figure 3.10(b), we show the reception distribution. We find that the distribution is even.



(a) (b)

Figure 3.9: Active radio time distribution of (a) *fc-Gappa* and (b) MNP. Inter-node distance: 10 feet. Program size: 14KB.



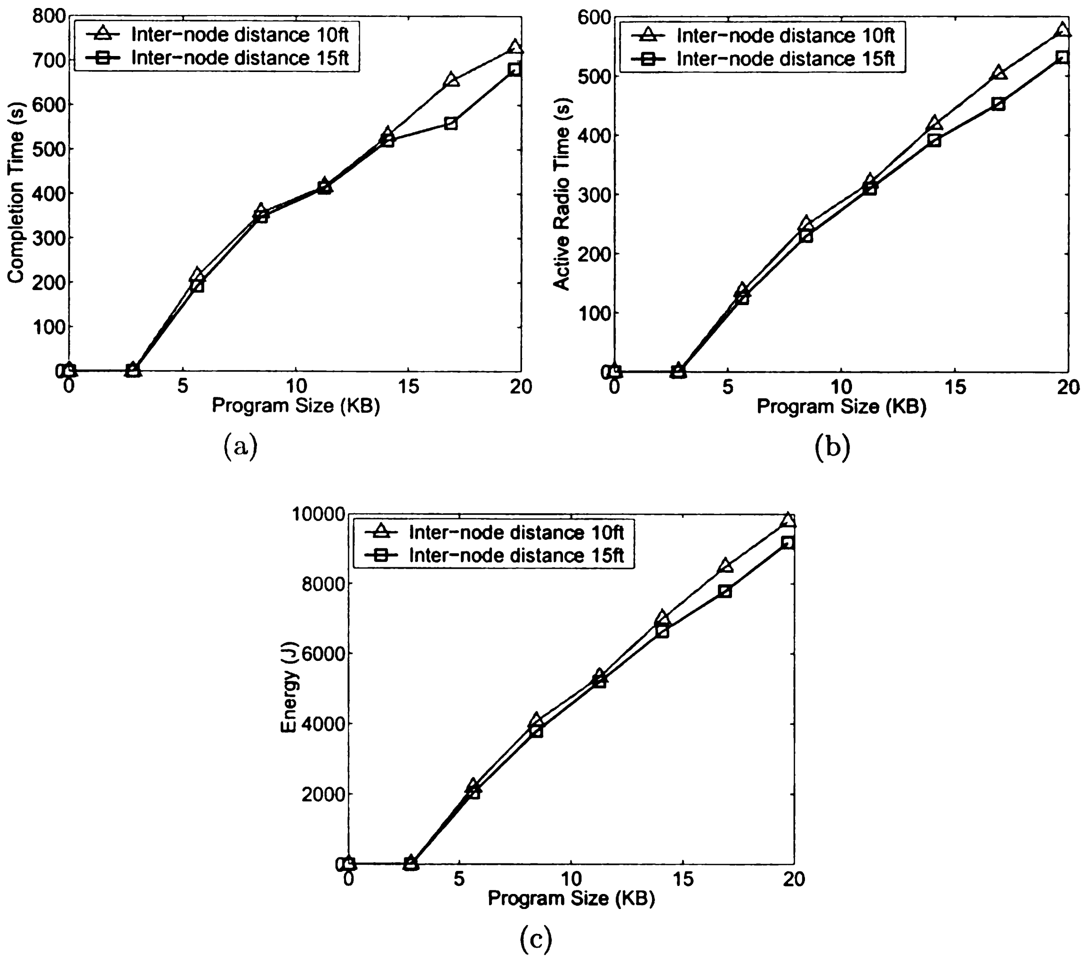
(a) (b)

Figure 3.10: *fc-Gappa*: message transmissions and receptions. Inter-node distance: 10 feet. (a) transmissions (b) receptions.

In Figure 3.11, we compare the performance of *fc-Gappa* at different node densities. We note that *fc-Gappa* performs well in both dense networks and sparse networks, although the performance in a sparse network is slightly better.

3.3.2 Varying Network Sizes

Simulation setup 5. In this section, we fix the inter-node distance to 10 feet and the program size to 5 segments (14.08KB, 640 packets), and conduct simulation



(a)

(b)

(c)

Figure 3.11: *fc-Gappa*: at inter-node distance 10 feet and 15 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

on different network sizes (10x10, 15x15, 20x20 grid). In Figure 3.12, we show that the completion time, average active radio time per node, average energy consumption per node increase slightly when the network size increases. For example, the completion time for reprogramming a 15x15 network with a 14KB program is 512 seconds, while the completion time for reprogramming a 20x20 network with a program of the same size is only 531 seconds; although the number of nodes almost doubles, the completion time only increases 3.6%. This shows that *fc-Gappa* scales well to large networks.

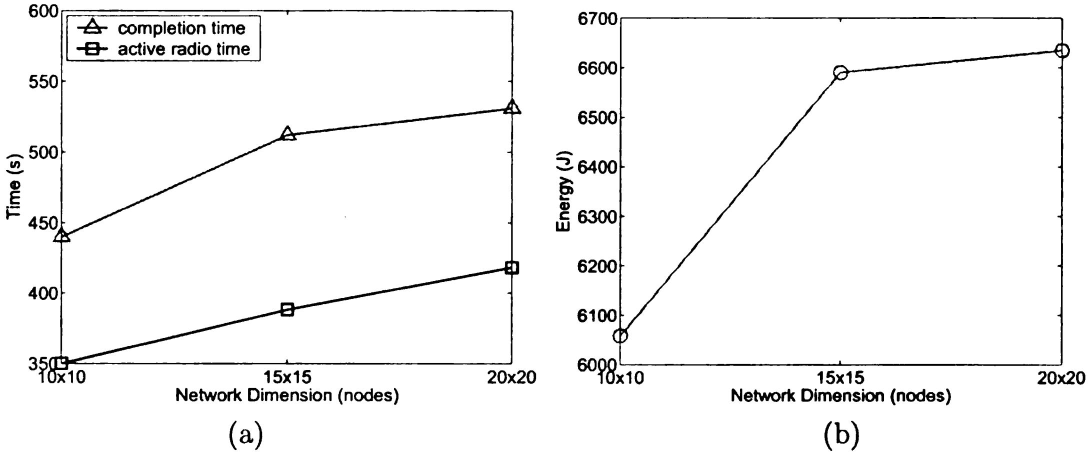


Figure 3.12: *fc-Gappa* at different network sizes. Inter-node distance: 10 feet. Program size: 14KB. (a) completion time and average active radio time per node (b) average energy consumption per node.

3.3.3 Varying Number of Seeds

Simulation setup 6. In this section, we study the situation where only a subset of nodes (seeds) have received a segment (randomly picked one) from an UAV initially. Each segment is received by at least one node. We conduct the simulation in a 20x20 network. The inter-node distance is set to 10 feet. The program size is 5 segments (14.08KB, 640 packets). We randomly select 5, 25, 50, 100, and 200 nodes as the seeds. The results are shown in Figure 3.13. For comparison, we also draw the corresponding results of MNP. We note that even if the nodes that have received one segment from the UAV are only 1.25% (each segment with exactly one node), *fc-Gappa* outperforms MNP in completion time. Additionally, from Figure 3.5 (a), it also outperforms Deluge.

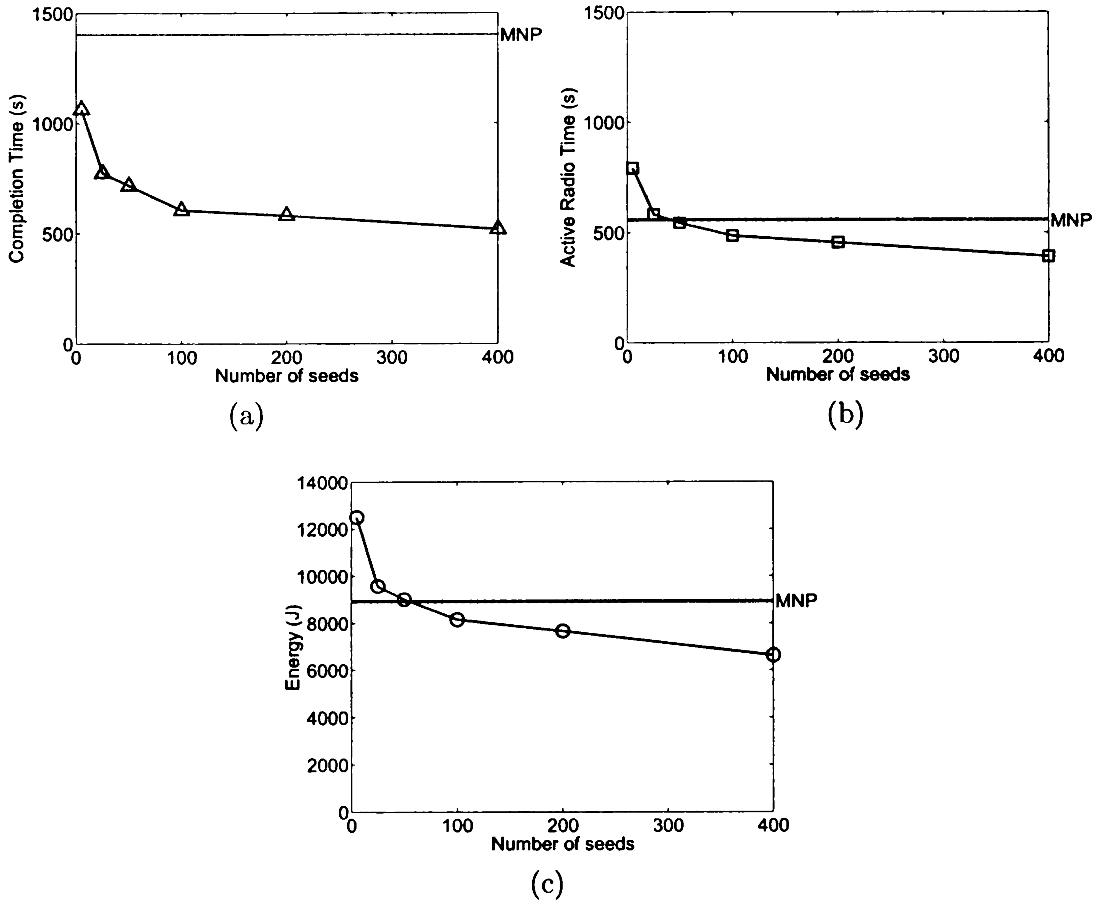


Figure 3.13: *fc-Gappa*: varying number of seeds. Inter-node distance: 10 feet. Program size: 14KB. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

3.4 Comparing Fixed Channel *Gappa* with Variable Channel *Gappa*

In Section 3.2, we presented *vc-Gappa*, which allows every sensor to randomly select a data channel to transmit data on. In this section, we evaluate the performance of *vc-Gappa*, and compare it with that of *fc-Gappa*. Corresponding to the simulations we performed on *fc-Gappa* in Section 3.3, we vary program size, network density, and network size, and show the comparison in Sections 3.4.1 and 3.4.2. In these two sections, we assume that all the sensors have received an initial segment that is randomly selected.

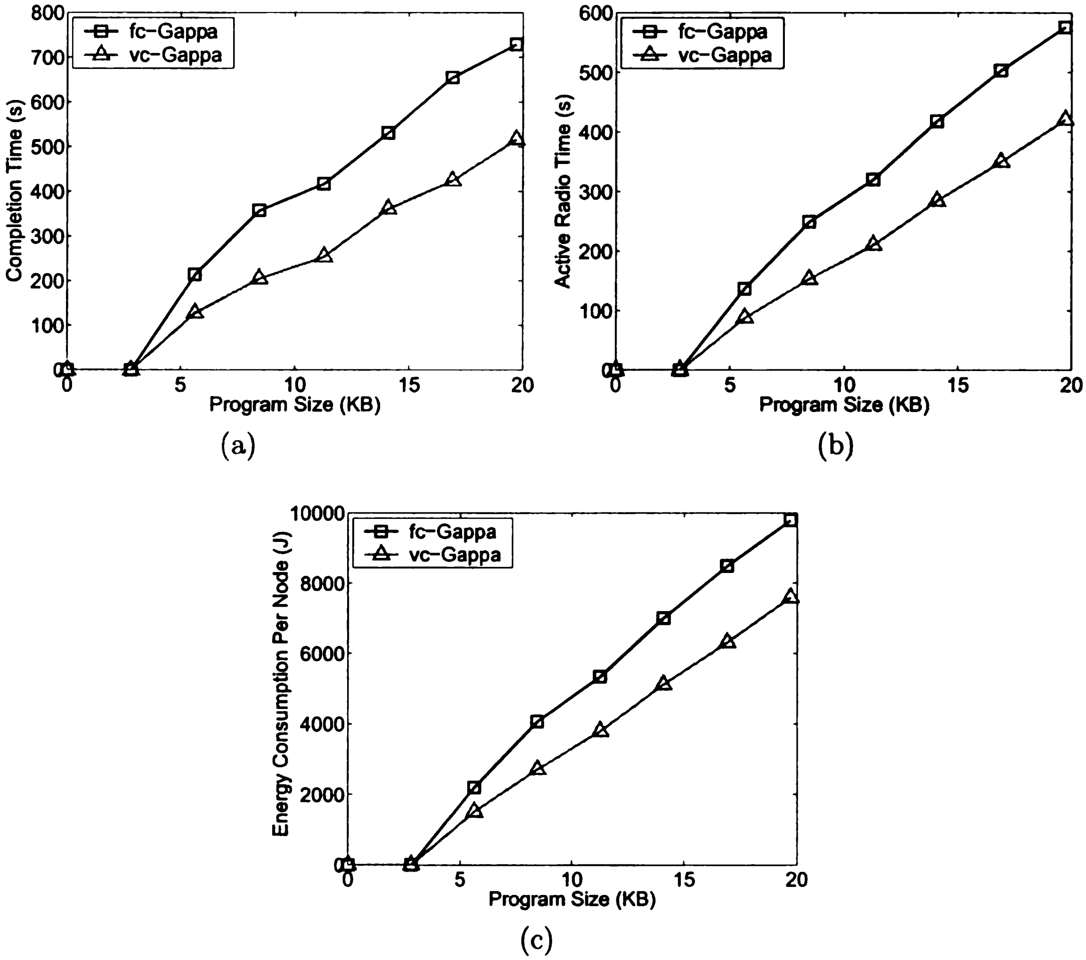
In Section 3.4.3, we consider the scenario where only one or two rows of sensors in the network receive a segment initially. Among the sensors that have initially received a segment, the first few columns receive the first segment, the next few columns receive the second segment, and so on. The intuition behind these simulations is that an UAV can be in contact with only a small number of segments at a time. During this time, it can transmit one segment. In all the simulations in this section, we assume that the total number of available channels is 54 (cf. Section 3.1).

3.4.1 Varying Program Sizes and Network Densities

Simulation setup 7. (*vc-Gappa*: dense network) We simulate *vc-Gappa* in a 20x20 network, and set the inter-node distance to 10 feet (a dense network). In Figure 3.14, we compare the completion time, the average active radio time per node, and the average energy consumption per node of *vc-Gappa* and *fc-Gappa*. We can see that with variable channel allocation, the performance of *Gappa* is effectively improved. Specifically, *vc-Gappa* reduces the completion time (respectively, the energy consumption) of *fc-Gappa* by 29-43% (respectively, 23-33%).

In Figure 3.15, we show the number of message transmissions and receptions per node of *fc-Gappa* and *vc-Gappa*. We note that in *vc-Gappa*, the number of transmissions (respectively, the number of receptions) per node is 13-20% (respectively, 18-29%) lower than that of *fc-Gappa*.

To illustrate the situation, we categorize the messages into two types, the data messages and the control messages. The data messages are those packets that contain the actual code. The remainings are called control messages, including advertisements, requests, "SwitchChannel" messages, "StartDownload" messages, etc. In Figure 3.16, we show the average number of data messages and control messages transmitted by a sensor under these two variations of *Gappa*. We can see that to reprogram



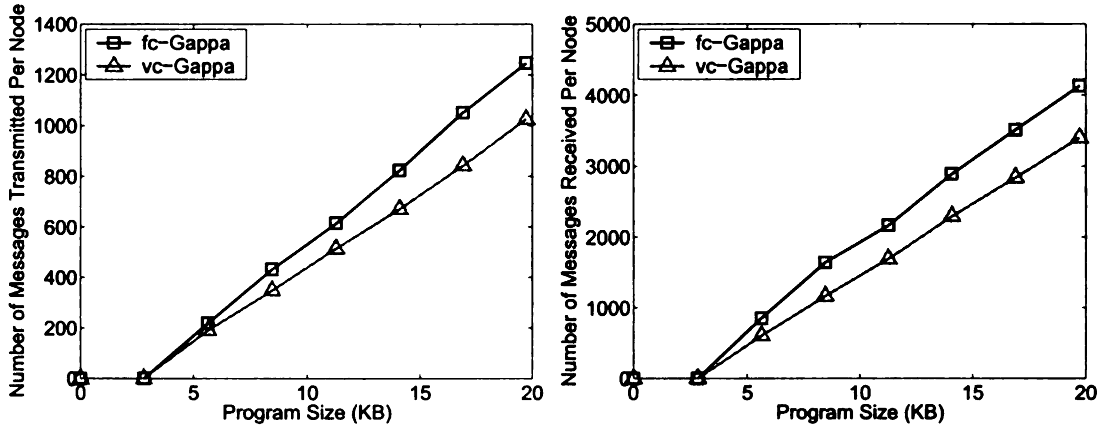
(a)

(b)

(c)

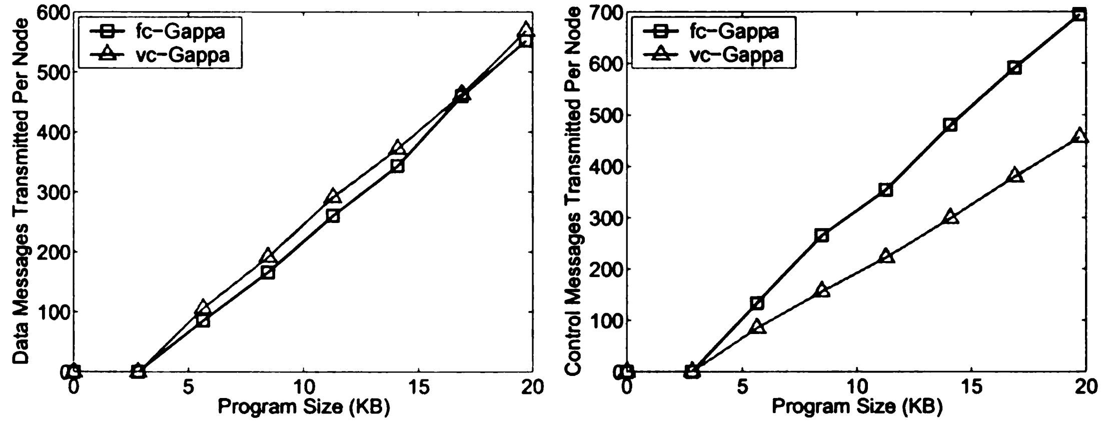
Figure 3.14: Comparison of *fc-Gappa* and *vc-Gappa*. Inter-node distance: 10 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

a network with a program of the same size, the number of data message transmissions of *vc-Gappa* is 3-24% higher than that of *fc-Gappa*, while the number of control mesage trnasmissions of *vc-Gappa* is about 40% lower than that of *fc-Gappa*. This is due to the fact that *vc-Gappa* allows higher concurrency during reprogramming, i.e., more sensors can transmit on multiple channels at the same time. With more concurrent senders, the average receiver set for each sender is smaller, which means more data packets needs to be sent. On the other hand, sensors are allowed to transmit as long as there are available channels, hence, they spend less time advertising and requesting. Thus, the control message transmission of *vc-Gappa* is significantly less than that of *fc-Gappa*.



(a) (b)

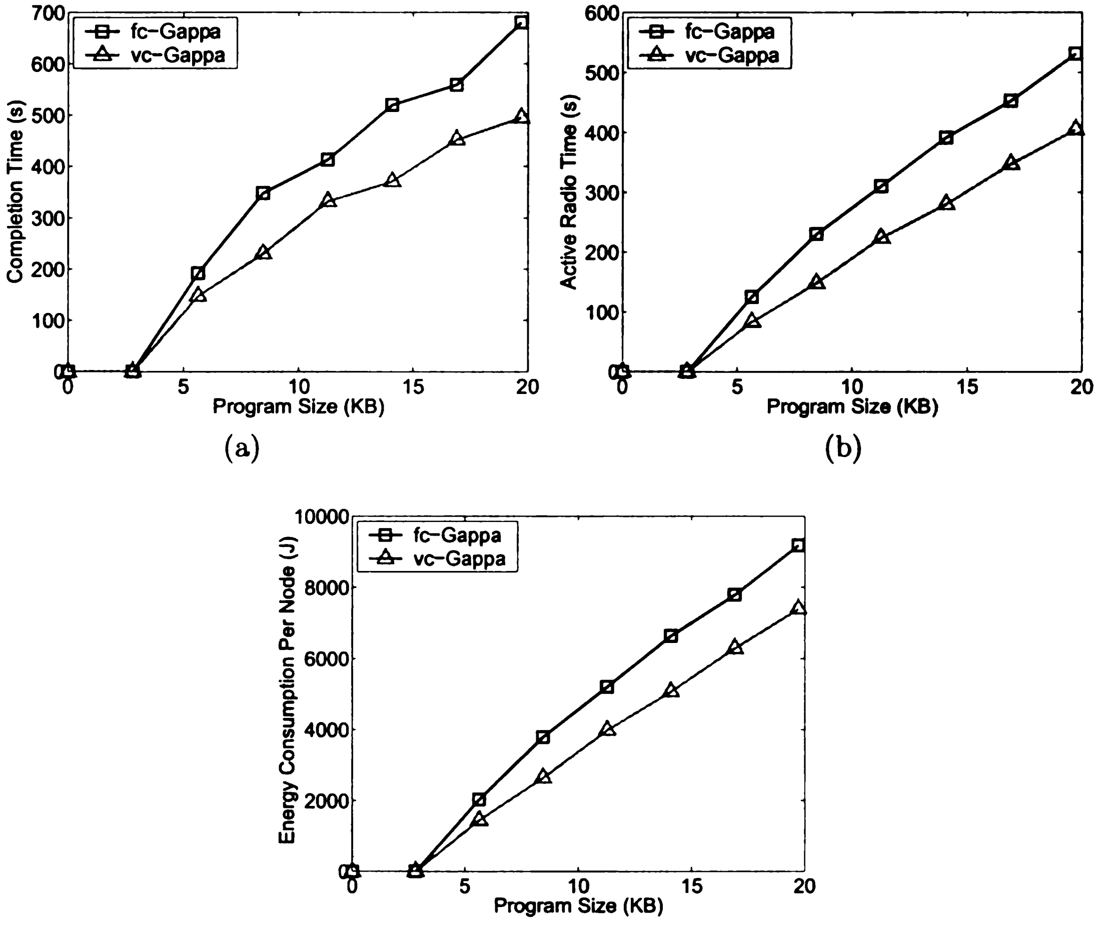
Figure 3.15: Comparison of *fc-Gappa* and *vc-Gappa*. Inter-node distance: 10 feet. (a) number of messages transmitted per node (b) number of messages received per node.



(a) (b)

Figure 3.16: Analysis of message transmissions in *fc-Gappa* and *vc-Gappa*. Inter-node distance: 10 feet. (a) number of *data* messages transmitted per node (b) number of *control* messages transmitted per node.

Simulation setup 8. (*vc-Gappa*: sparse network) We repeat the same set of simulations in a sparse network where the inter-node distance is 15 feet. We show the completion time, the average active radio time per node and the average energy consumption per node in Figure 3.17. Similar to the dense network case, *vc-Gappa* reduces the completion time (respectively, energy consumption) of *fc-Gappa* by 19-34% (respectively, 19-30%). In Figure 3.18, we can see that *vc-Gappa* has lower message transmissions and receptions than *fc-Gappa*.



(a)

(b)

(c)

Figure 3.17: Comparison of *fc-Gappa* and *vc-Gappa*. Inter-node distance: 15 feet. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

3.4.2 Varying Network Sizes

Simulation setup 9. In this section, we set the inter-node distance to 10 feet and the program size to 5 segments (14.08 KB) and vary the network size from 10x10 to 20x20. In Figure 3.19, we can see that the completion time (respectively, the

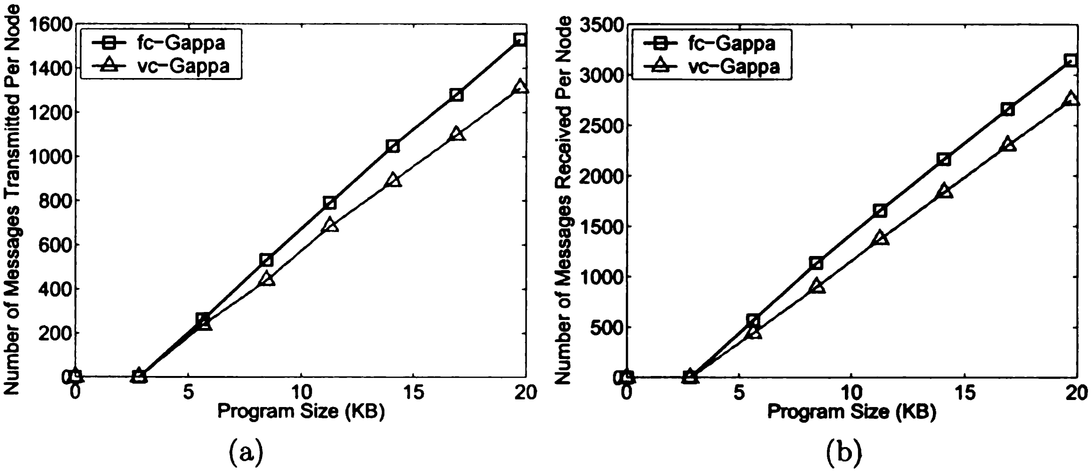


Figure 3.18: Comparison of *fc-Gappa* and *vc-Gappa*. Inter-node distance: 15 feet. (a) number of messages transmitted per node (b) number of messages received per node.

average energy consumption per node) of *vc-Gappa* is only around 68% (respectively, 73%) that of *fc-Gappa*.

3.4.3 Varying Initial Distribution of Segments

Simulation setup 10. In this section, we focus on the case where the two rows of sensors in the center of a 20x20 network receive a code segment from an UAV directly. Moreover, we assume that the speed of the UAV is almost constant, i.e., the number of sensors that receive each segment is about the same. For example, if the program has 5 segments, then the first 4 columns (in the center two rows) receive segment 1, the next 4 columns (in the center two rows) receive segment 2, and so on. In the case that the program has 3 segments, the first 7 columns (in the center two rows) receive segment 1, the next 6 columns (in the center two rows) receive segment 2, the last 7 columns (in the center two rows) receive segment 3. In our simulation, we set the program size to 2, 3, 5 segments, respectively. The inter-node distance is 10 feet.

We simulate *fc-Gappa* and *vc-Gappa* in this setting. In Figure 3.20, we show the completion time, the average active radio time per node, and the average energy

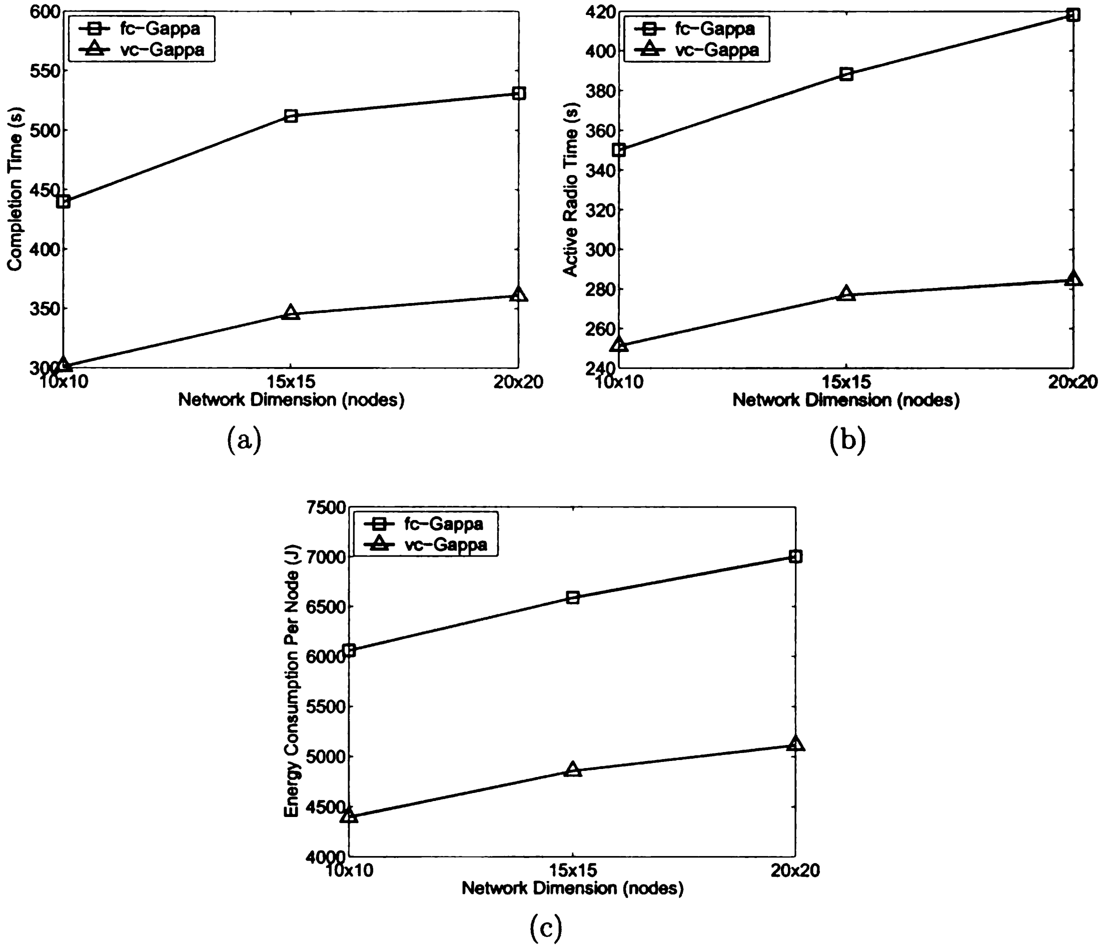
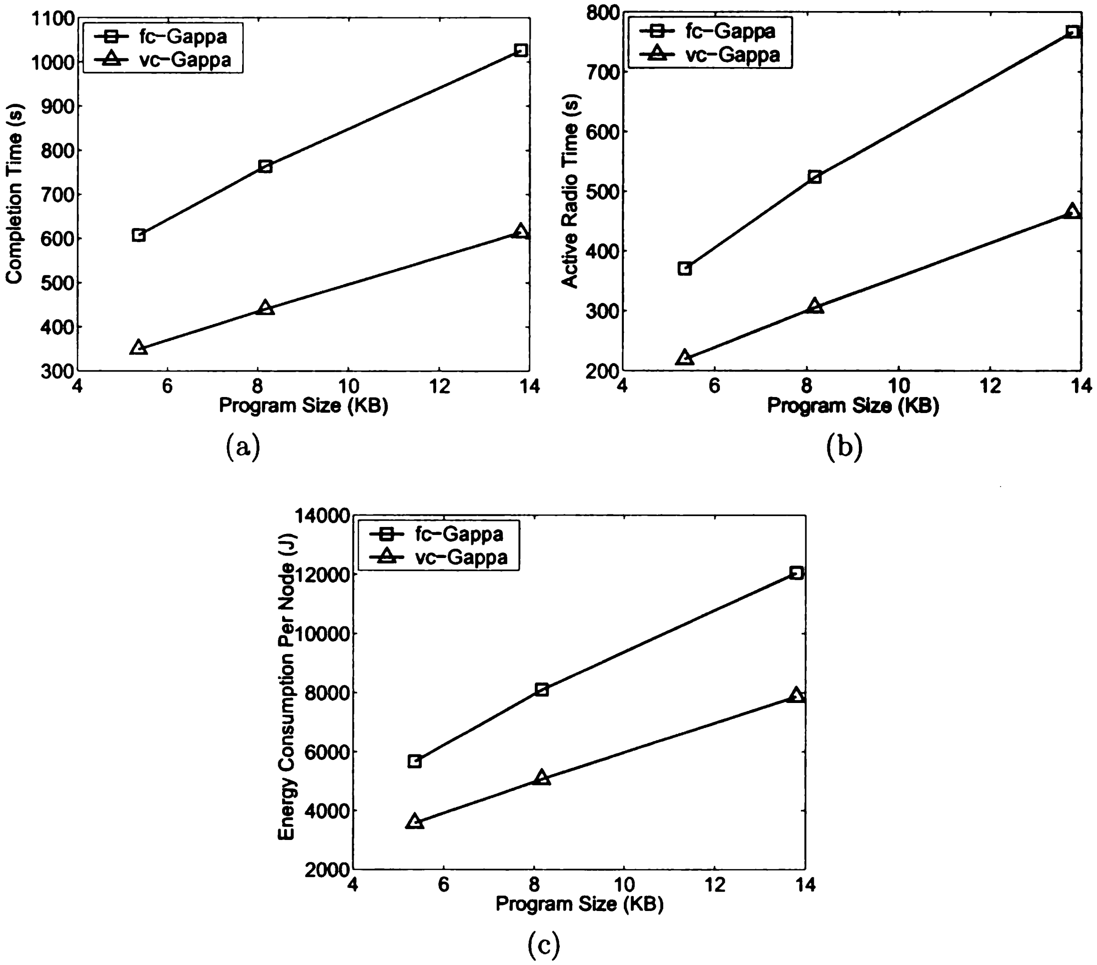


Figure 3.19: Comparison of *fc-Gappa* and *vc-Gappa* at different network sizes. Inter-node distance: 10 feet. Program size: 14KB. (a) completion time (b) average active radio time per node (b) average energy consumption per node.

consumption per node of these two protocols. We note that in this case, due to the fact that the sensors that receive the same segments initially are physically close to each other (as opposed to random distribution in the previous setting), the variable channel allocation scheme used in *vc-Gappa* outperforms the fixed channel allocation scheme significantly. We can see that *vc-Gappa* reduces the completion time (respectively, the energy consumption) of *fc-Gappa* by more than 40% (respectively, more than 35%). The communication overhead of *vc-Gappa*, as shown in Figure 3.21, is a little lower than that of *fc-Gappa*.



(a)

(b)

(c)

Figure 3.20: Comparison of *fc-Gappa* and *vc-Gappa* in the case where the UAV broadcasts code segments on a single channel and has a small receiver set. Inter-node distance: 10 feet. The program size is 2, 3, 5 segments. The center two lines of sensors receive a code segment from the UAV initially. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

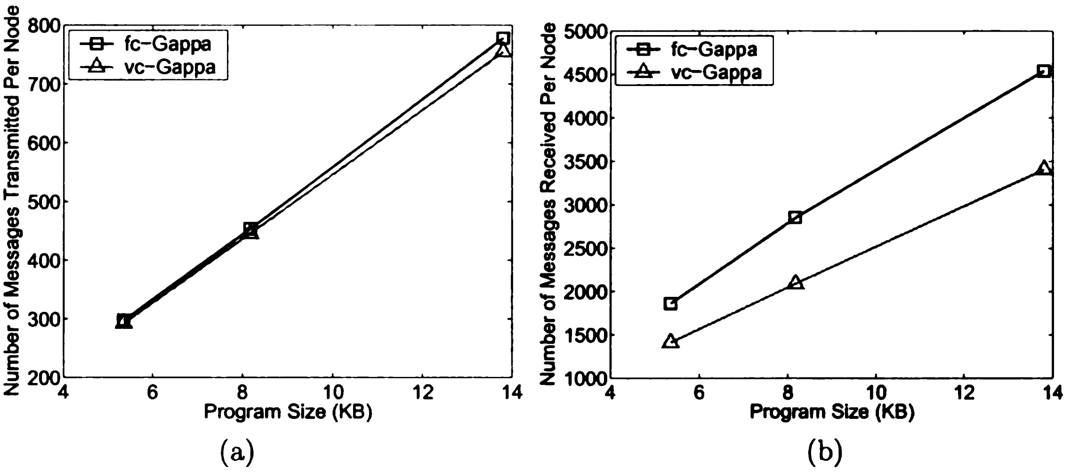
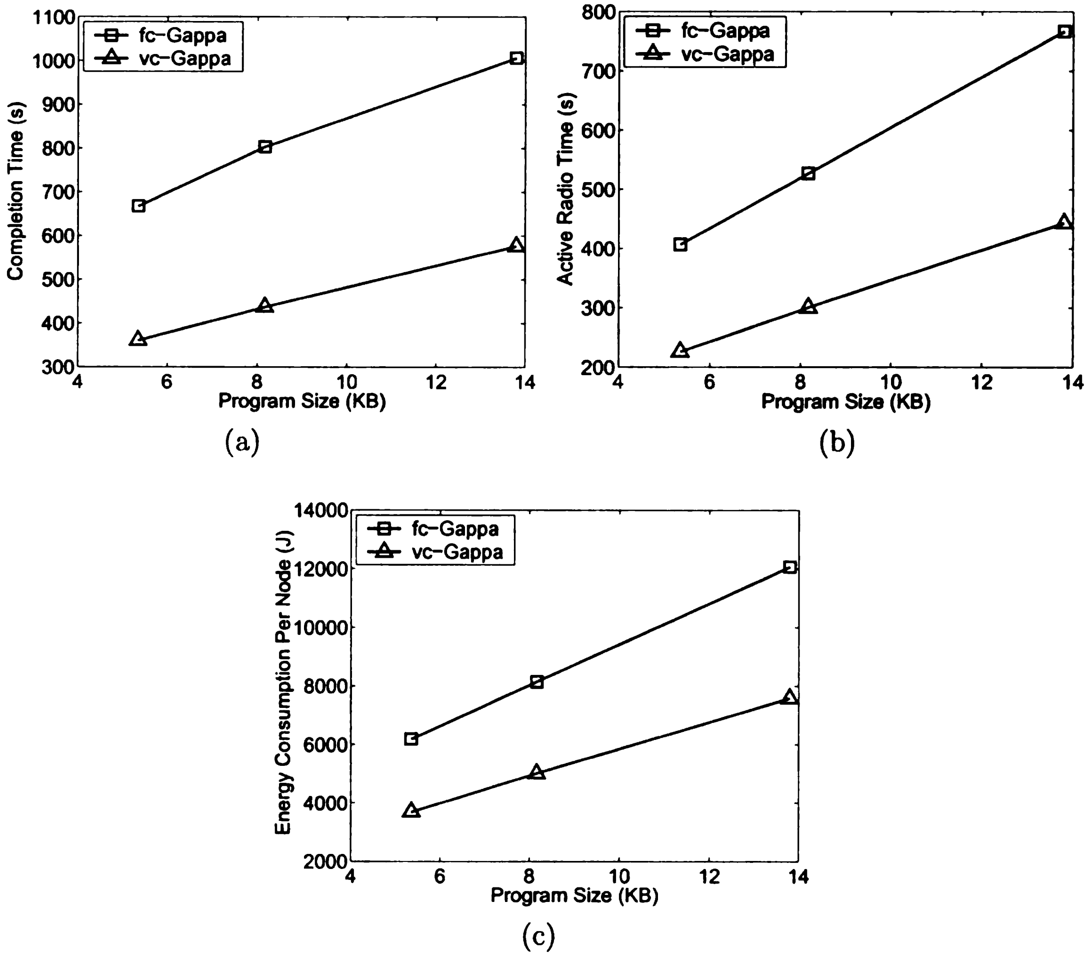


Figure 3.21: Comparison of *fc-Gappa* and *vc-Gappa* in the case where the UAV broadcasts code segments on a single channel and has a small receiver set. Inter-node distance: 10 feet. The program size is 2, 3, 5 segments. The center two lines of sensors receive a code segment from the UAV initially. (a) number of messages transmitted per node (b) number of messages received per node.

Simulation setup 11. In this simulation, we change the initial setting in Simulation setup 10 in such way that only one row of sensors in the center of the network receive a segment. We show the corresponding results in Figures 3.22 and 3.23. We can see that the results are similar to those we have shown in Figures 3.20 and 3.21 for Experiment setup 10, i.e., *vc-Gappa* outperforms *fc-Gappa* significantly in terms of completion time and energy consumption. And, *vc-Gappa* has a little lower communication cost than *fc-Gappa*.

3.5 Chapter Summary

In this chapter, we presented *Gappa*, a gossip based multi-channel reprogramming protocol for sensor networks, that is designed for the scenario where some sensor nodes receive one part of the new program initially, and communicate with each other so that all the nodes in the network receive the entire program after reprogramming. By exploiting multi-channel resources and pipelining technique, *Gappa* enables high



(a)

(b)

(c)

Figure 3.22: Comparison of *fc-Gappa* and *vc-Gappa* in the case where the UAV broadcasts code segments on a single channel and has a small receiver set. Inter-node distance: 10 feet. The program size is 2, 3, 5 segments. The line of sensors at the center of the network receive a code segment from the UAV initially. (a) completion time (b) average active radio time per node (c) average energy consumption per node.

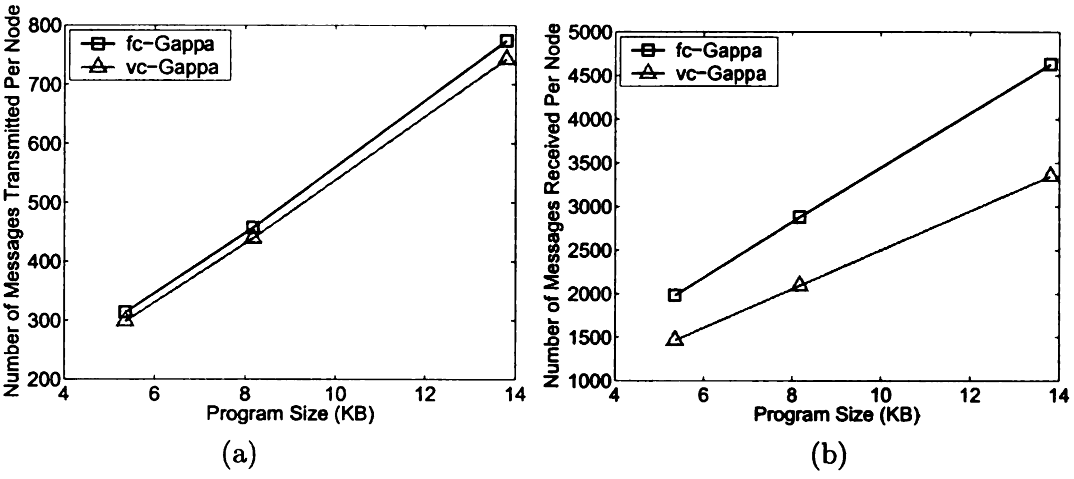


Figure 3.23: Comparison of *fc-Gappa* and *vc-Gappa* in the case where the UAV broadcasts code segments on a single channel and has a small receiver set. Inter-node distance: 10 feet. The program size is 2, 3, 5 segments. The line of sensors at the center of the network receive a code segment from the UAV initially. (a) number of messages transmitted per node (b) number of messages received per node.

concurrency on different channels and different locations, hence, propagates data rapidly. To reduce collisions on each channel, *Gappa* uses a multi-channel sender selection algorithm (based on the sender selection algorithm in MNP), which tries to guarantee that at any neighborhood, at most one sender transmits on one channel at a time. Among the competing senders on each channel, the multi-channel sender selection algorithm attempts to select the one whose transmission of the program on that channel is likely to have the most impact. In the case that a node loses the sender selection on one channel, it has the option to compete to transmit on another channel. If all the channels a node can transmit code on are busy, the node stops advertising for a certain amount of time. During that time, it can choose, based on its status, to wait to receive code with its radio on, or to turn off radio to save energy. In this way, *Gappa* reduces the active radio time of sensor nodes, hence, energy consumption, during reprogramming.

We present two variations of *Gappa*, *fc-Gappa* and *vc-Gappa*, that use fixed channel allocation and variable channel allocation, respectively. In *fc-Gappa*, each segment is assigned one channel, while in *vc-Gappa*, the sensors select a channel randomly from all the available channels.

We evaluated *fc-Gappa* through simulation on TOSSIM, and compared it with the other two state-of-art reprogramming protocols, MNP and Deluge, both of which assume that initially only the base station(s) has the entire program. The simulation results show that under the same network settings, to propagate a program of the same size, *fc-Gappa* saves up to 70% of completion time and up to 42% energy consumption compared to MNP, and saves up to 84% completion time and up to 88% energy consumption compared to Deluge. We also show that *fc-Gappa* adapts well to different network densities and network sizes. Moreover, we note that *fc-Gappa* distributes energy load more evenly. This is expected to help in maintaining a longer network lifetime.

We also considered the case where only a subset of nodes receive a segment of code initially. In the simulation, we study the worst cases where only 1.25% to 50% of nodes have received a part of the code. The simulations results show that even in these situations, *fc-Gappa* still outperforms MNP and Deluge in completion time.

We note that *vc-Gappa*, which uses a little more complicated channel allocation scheme, performs even better than *fc-Gappa* in all the network settings. The variable channel allocation scheme used in *vc-Gappa* allows neighboring sensors to transmit data simultaneously as long as there are available channels. For this reason, it is especially beneficial in the cases where the network is dense or the initial distribution of code segments is not random.

Chapter 4

Proactive Reliable Data Dissemination

Network reprogramming requires 100 percent delivery of the entire binary image (on the order of kilobytes), and hence consumes significant communication bandwidth. However, the transmissions are performed over radio, which is known as a low-bandwidth and lossy medium. Therefore the reliability issues need to be addressed.

There are two basic methods to recover lost packets. One way is to use automatic repeat request (ARQ). In ARQ schemes, a receiver detects its own losses, and informs the sender of the missing packets, either by sending requests (NACK) or acknowledgments (pure ACK, implicit ACK, selective ACK, etc.). The sender retransmits the repair packet if it knows that a packet is lost. Another way to recover errors is to use forward error correction (FEC). FEC provides reliability by transmitting redundant packets in a proactive manner. The most commonly used FEC scheme is (n, k) FEC. The fundamental of (n, k) FEC is to add $n - k$ additional packets to a group of k source packets (called a *transmission group*) so that the receipt of any k packets at the receiver permits recovery of the original k ones. There are different levels of FEC schemes: packet-level, byte-level, bit-level. In the context of reprogramming, we only examine the packet-level FEC.

The existing protocols on network reprogramming include the single-hop reprogramming protocol XNP [12], and multihop reprogramming protocols MOAP [60], Deluge [23], MNP (cf. Chapter 2), Infuse [30], and Sprinkler [47]. All these protocols use automatic repeat request (ARQ) scheme to recover from packet losses. ARQ is an effective reliability scheme, as an error can always be recovered as long as the network is connected. However, if the error rate is high, the requests and retransmissions for the missing packets consume significant energy. In this chapter, we examine the issue of adding FEC to the ARQ-based reliability scheme of a reprogramming protocol. We perform a case study on MNP. The proposed reliability scheme is a hybrid approach of FEC and ARQ. By adding FEC, we expect to reduce the error probability experienced at the receivers, and ARQ scheme performs the remaining error corrections.

In Section 4.1, we study the packet loss pattern in MNP. In Section 4.2, we propose a hybrid reliability scheme using FEC and ARQ, and describe the implementation details of adding simple XOR code and Reed-Solomon (RS) codes [55] to MNP. In Section 4.3, we present the simulation results on the performance of (MNP+XOR) and (MNP+RS codes). We summarize this chapter in Section 4.4.

4.1 Packet Loss Pattern in MNP

Before adding FEC, we ran a simulation to see the packet loss pattern, which is shown in Figure 4.1. The simulation was conducted in a 10x10 network with 10 feet inter-node distance. The program size is 8.4KB (3 segments, 384 packets). The x-axis is the number of missing packets indicated by the *MissingVector* contained in a request message. The y-axis is the number of request messages. Since the segment size is 128 packets, the number of missing packets ranges from 1 to 128 packets. The peak at the right is the number of requests that ask for the whole segment (128 packets). These are *protocol requests*, used in the sender selection algorithm. We only consider *repair requests*, the requests that ask for *less than* 128 packets. We note that most of the repair requests only ask for a few number of packets. For example, about 50% of the requests are asking for less than 8 missing packets, 70% of the requests are asking for less than 16 missing packets, 83% of the requests are asking for less than 32 missing packets. The fact that the majority of the losses are small losses (involving a few number of missing packets) suggests that a better link is desirable to reduce the number of requests and retransmissions. FEC can be used to provide an abstraction of an enhanced link at the cost of transmitting additional parity packets.

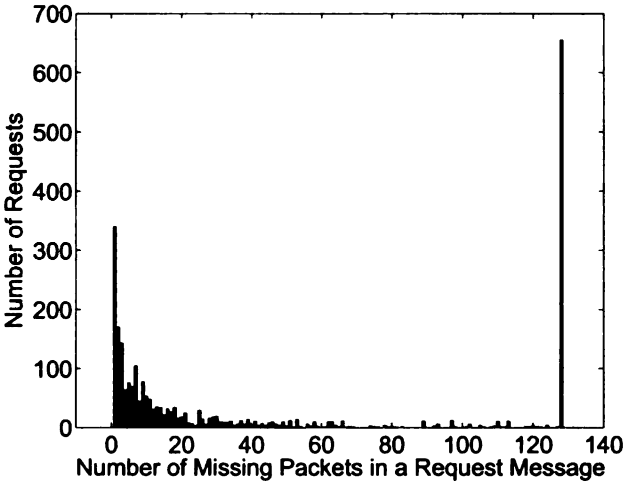


Figure 4.1: Packet loss pattern in MNP. 10x10 network, inter-node distance: 10 feet, program size: 8.4KB (384 packets).

4.2 A Hybrid Reliability Scheme for MNP

In Section 4.2.1, we briefly introduce the two (n, k) FEC coding schemes: simple XOR code and Reed-Solomon (RS) codes. In Section 4.2.2, we present the new reliability scheme for MNP, that is, adding XOR/RS FEC codes to MNP.

4.2.1 (n, k) FEC Coding Schemes

We consider (n, k) FEC-based approaches. There are two commonly used (n, k) FEC codes: XOR code, and Reed-Solomon (RS) codes [55]. For simple XOR code, each transmission group has only one parity packet, which is the XOR of all the source packets in the group. Therefore, simple XOR code is a $(k + 1, k)$ code. XOR code is very simple to implement. However, it can only repair a single packet loss in a transmission group.

RS codes are more flexible. RS codes are based on algebraic methods using finite fields. A transmission group can have multiple parity packets (i.e., n can be any number that is larger than k). Thus RS codes provide better protection against losses. However, the flexibility of RS codes is achieved at high processing costs, in terms of computation complexity and memory space.

4.2.2 Adding FEC to MNP

There are two approaches regarding the calculation of the parity packets. The first approach is to calculate the parity packets based on the actual packets that are sent in the current transmission. In this case, for every transmission, the parity packets are computed and sent. This incurs a lot of encoding and decoding overhead. Moreover, the receivers have to be aware of the *ForwardVector* of the current sender (cf. Section 4.1), that is, the packets that are sent in the current transmission, in order to decode the parity packets. The sender might need to send the *ForwardVector* several times in order to make sure that it is received by all the receivers. The second approach is to compute the parity packets based on the full segment. Compared to the first approach, this approach is more efficient because the encoding process is needed only once for each segment, rather than performed for each transmission; and the senders do not need to send *ForwardVector* to the receivers. Therefore, we use the second approach.

For each segment, there are a set of parity packets. The parity packets are assigned unique IDs that start from *SegSize*+1. A receiver keeps a bitmap of the parity packets, *ParityMissingVector*, in memory. The *ParityMissingVector* is operated just as the *MissingVector* in the original ARQ-based approach (cf. Section 4.1). All the bits in *ParityMissingVector* are set to 1 initially. When a node receives a parity packet for the first time, it set the corresponding bit to 0. We limit the size of the *ParityMissingVector* to be no larger than 4 bytes, thus the number of parity packets ranges from 1 to 32.

When a receiver sends a request, it puts *MissingVector*, as well as *ParityMissingVector*, in the request message. Correspondingly, a source node maintains a *ParityForwardVector* (in addition to the *ForwardVector* in the original algorithm), as an indicator of the *parity* packets that need to be sent if the node becomes a sender. The *ParityForwardVector* of a source node is the union of the *ParityMissingVector* in the request messages the node has received. Whenever a receiver receives a packet (either data or parity), it checks to see if the number of packets it has received is enough to recover all the missing data packets. If so, the receiver has received the entire segment, otherwise, it asks for the missing data and parity packets, and a sender sends the requested packets.

The difference between adding XOR code and Reed-Solomon codes is the capability of loss recovery and the complexity of encoding and decoding, as we mentioned in Section 4.2.1. Because XOR code can only have one parity packet and repair a single loss in a transmission group, in order to repair more than one loss in a segment, we divide a segment into t transmission groups. Each group has a fixed number ($SegSize/t$) of data packets and one parity packet. The parity packet that has ID *SegSize*+i ($1 \leq i \leq t$) corresponds to the i^{th} transmission group. In each transmission, the sender sends all the requested data packets, followed by the requested parity packets. Whenever a node has received enough number of packets to recover the whole group, it sets all the bits of this group in *MissingVector* and *ParityMissingVector* to 0, so that the node will not request for packets within this group any more.

For Reed-Solomon codes, we consider a full segment as a transmission group, which can have multiple parity packets. As long as the receiver has received *SegSize* (or more) number of packets (either data packets or parity packets), the whole segment is received. Further optimization is possible. For example, consider one scenario where a receiver has 11 missing data packets, and has received 8 parity packets. In this case, the parity packets received are not enough to recover the losses. In order to recover the whole segment, the receiver only needs 3 more packets, rather than 11. Taking message losses into account, we allow the receivers to request for twice the *required* number of missing packets (k packets in (n, k) FEC schemes). In the previous example, the receiver will request for $2 \times 3 = 6$ packets. This feature is expected to reduce the number of retransmissions, especially when the number of parity packets is large. However, because in MNP, the packets a sender transmits is the combination (union) of the packets that are requested, it is likely that different receivers request for different sets of packets, thus the combination of them virtually covers the whole segment. In this case, this optimization is not effective. We add one restriction that if a node requests for a subset of the packets it is missing, it always requests for those packets that have the lowest IDs, so that it is more possible that the sets of packets requested by different receivers are overlapping.

4.3 Evaluation Results

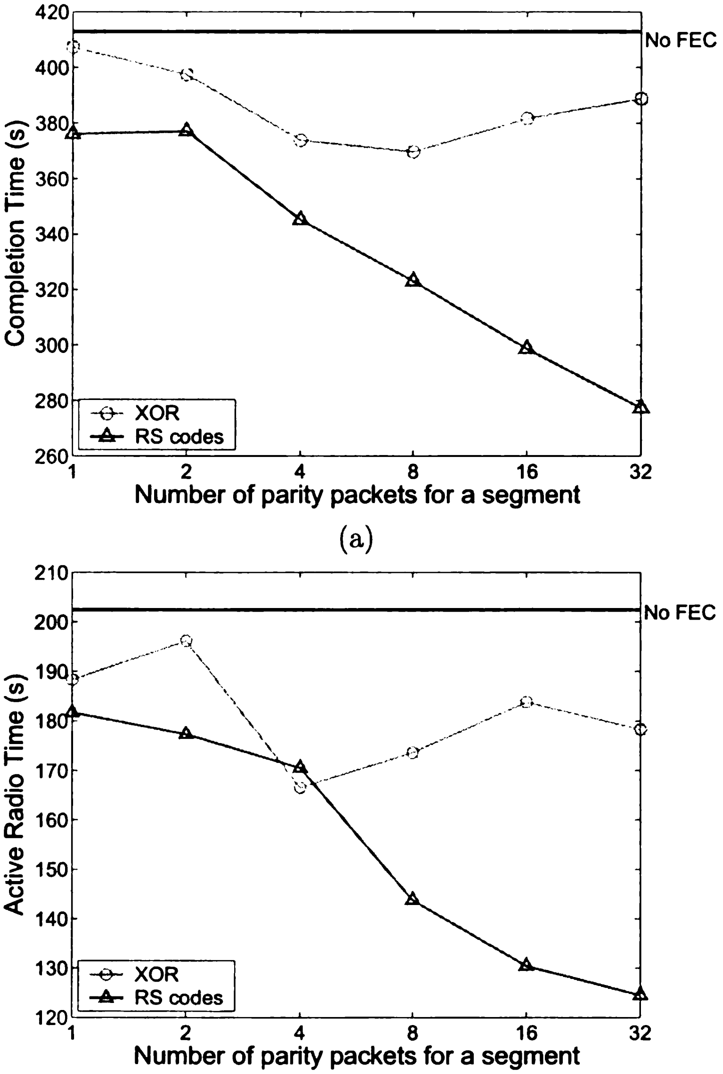
We added simple XOR code and RS codes to MNP source code, as described in Section 4.2.2, and used TOSSIM, to evaluate the effectiveness of the new reliability scheme.

The following simulations were conducted in a 10x10 network. The distance between two neighboring nodes is kept constant at 10 feet. In the current implementation, each segment has 128 data packets. The program size is 8.4KB(3 segments, 384 packets). We assume that initially only the base station, the node at a corner, has the new program.

In Figure 4.2, we compare the performance of the original MNP protocol (marked as "No FEC" in Figure 4.2) with MNP protocol plus XOR/RS FEC codes. To prevent the randomness of a single simulation, we repeated each simulation three times, and presented the mean value. We found that adding either simple XOR code or RS codes to MNP helps improving performance. The completion time and the active radio time were reduced after we applied FEC schemes to MNP. For XOR code, when the number of parity packets is from 4 to 8, the reduction on completion time is about 10%, and the reduction on active radio time is about 14-17%. Increasing or reducing the number of parity packets minuses the performance gain. RS codes generally perform better than simple XOR code. As shown in Figure 4.2, using RS codes, the completion time is reduced by 9-33%, and the active radio time is reduced by 10-38%. For RS codes, the performance improves when more parity packets are used.

In Figure 4.3, we show the number of transmissions and receptions of the original MNP protocol and MNP plus XOR/RS FEC schemes. We note that, with XOR/RS FEC schemes, the number of transmissions and receptions are reduced by up to 19% and 41% respectively. In general, (MNP + XOR) scheme has lower number of transmissions and receptions than the original MNP protocol, and (MNP + RS codes) has the lowest number of transmissions and receptions among the three schemes. The number of receptions is largely decided by the average active radio time, as can be seen from Figure 4.2(b) and Figure 4.3(b).

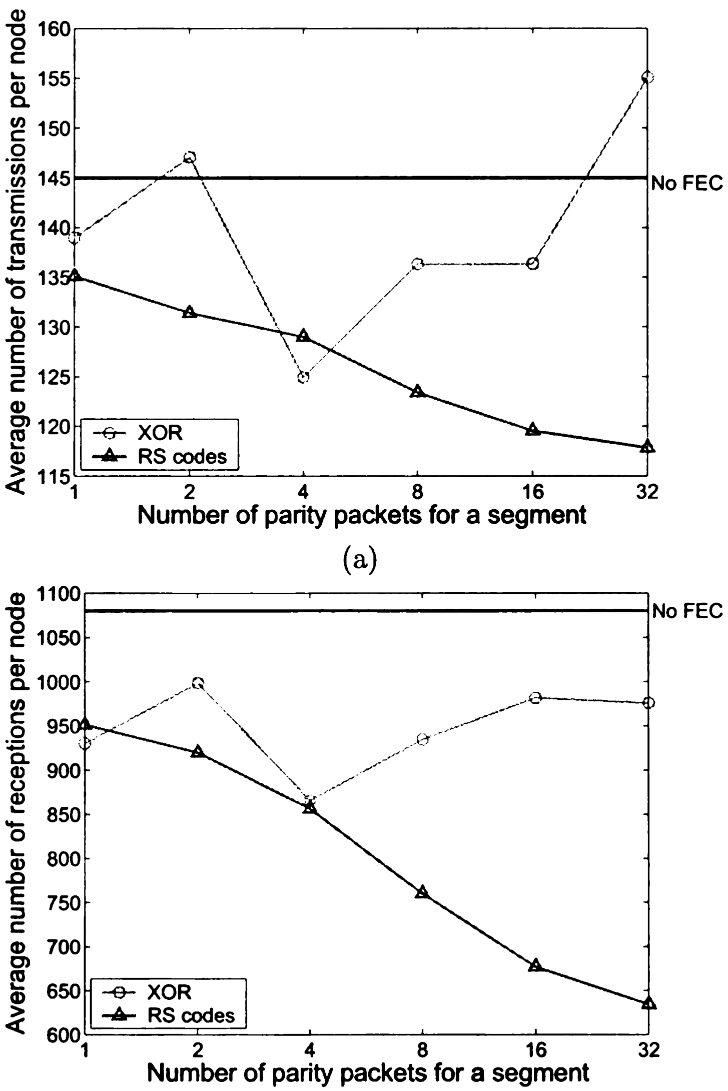
We categorize the transmitted packets into two types: control packets and en-



(a)

(b)

Figure 4.2: Completion time and active radio time of (MNP + XOR) and (MNP + RS codes), when the number of parity packets is from 1 to 32 packets per segment (128 data packets/segment). (a) Completion time (b) Average active radio time per node.

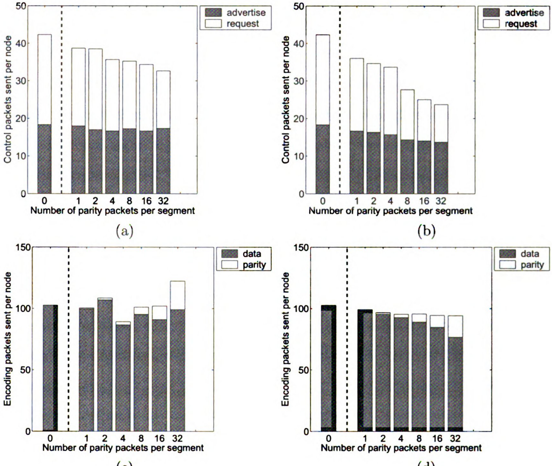


(a)

(b)

Figure 4.3: Average number of transmissions and receptions per node: (MNP + XOR) and (MNP + RS codes), when the number of parity packets is from 1 to 32 packets per segment (128 data packets/segment). (a) Average number of transmissions per node (b) Average number of receptions per node.

coding packets. Control packets include the advertisements and requests. They are used for the sender selection algorithm and ARQ scheme. Encoding packets carry the information that is to be disseminated to the sensor nodes. They include data packets and parity packets. In Figure 4.4 (a) and (b), we show the average number of control packets that are transmitted per node, for (MNP + XOR) scheme, and (MNP + RS codes) scheme, compared to the original MNP protocol. In Figure 4.4 (c) and (d), we show the corresponding results for encoding packets.



(a) (b) (c) (d)

Figure 4.4: The average number of control/encoding packets transmitted per node. (a) control packets transmitted in (MNP + XOR) scheme (b) control packets transmitted in (MNP + RS codes) scheme (c) encoding packets transmitted in (MNP + XOR) scheme (d) encoding packets transmitted in (MNP + RS codes) scheme. * 0 - means original MNP (No FEC/parity)

We note that using XOR/RS codes effectively reduces the number of control packets (up to 44%), especially the number of requests. For example, when we use RS codes with 32 parity packets per segment, the average number of requests per node is only 10 (Figure 4.4(b)), less than half of the requests transmitted per node when the original MNP is used. As to encoding packets, we note that although FEC schemes transmit additional parity packets, the number of data packets, plus the parity packets, is still lower than the number of data packets transmitted by the original MNP algorithm in general, with a few exceptions (when the number of parity packets per segment is 2 or 32 in (MNP + XOR) scheme).

From Figure 4.4, we can see how RS codes performance better than simple XOR code. For XOR code, when the number of parity packets per segment increases from 4 to 32, the additional parity packets do not contribute much to recovering packet losses, but incur higher transmission overhead due to more redundant packets. As we mentioned in Section 4.2, the limitation of XOR scheme is that, only one loss can be recovered in each transmission group. Although we can use multiple parity packets for a segment by dividing a segment into several transmission groups, only one loss from each group can be recovered. In other words, XOR code with multiple parity packets work best when the message losses are evenly distributed in the segment. However, in network reprogramming, a large part of the message losses are bursty in nature, caused by message collisions or channel errors. Therefore, dividing a segment into many tiny groups, and transmitting one parity packet for each group using XOR code, is not desirable. By contrast, RS codes can recover message losses at arbitrary locations. For RS codes, increasing the number of parity packets in a group improves its ability of recovering message losses. As shown in Figure 4.4 (c) and (d), for XOR code, when more than 4 parity packets are used for each segment, the number of encoding packets increases with the increase of the number of parity packets; for RS codes, the number of encoding packets remains the same although more parity packets are transmitted.

4.4 Chapter Summary

Automatic repeat request (ARQ) is a commonly used technique for reliable bulk data dissemination applications in sensor networks. A typical example of this type of applications is network reprogramming. All the existing network reprogramming protocols use ARQ as the error recovery scheme. In this chapter, we proposed a hybrid reliability scheme, which combines forward error correction (FEC) with ARQ schemes. The FEC provides an abstraction of a better transmission medium, and ARQ scheme takes care of the remaining error corrections. We use MNP, a multihop network reprogramming protocol, as a study case, and presented the implementation details of adding FEC schemes to MNP. Specifically, we considered two (n, k) FEC schemes: the simple XOR code, and Reed-Solomon (RS) codes. We added the two FEC schemes to MNP, and simulated them in TOSSIM.

The simulation results show that both simple XOR code and Reed-Solomon codes effectively reduce the number of request messages that ask for missing packets, thus enable faster reprogramming and less energy consumption. Adding XOR code to MNP can reduce the reprogramming time by 1-10%, and reduce the active radio time (which contributes to the major part of energy consumption) by 3-18%; while adding Reed-Solomon codes to MNP can reduce the reprogramming time by 9-33%, and reduce the active radio time by 10-38%. The message transmissions and receptions are reduced as well when the FEC schemes are used. We found that simple XOR code has limited capability of correcting errors (reducing completion time by up to 10% and reducing active radio time by up to 18%), that is, it cannot deal with bursty packet losses, which is not rare when disseminating a large amount of data in large networks. Therefore, increasing the number of parity packets does not help to improve the performance, only incurs more redundant transmissions. On the other hand, Reed-Solomon codes are more flexible. It can deal with any loss patterns. It becomes more powerful at correcting errors when more parity packets are used. XOR code is very simple to implement, while Reed-Solomon codes require higher computing resources due to the complexity of their calculation. It suggests a tradeoff between computation and performance: if more computation resources are available, a more powerful coding scheme, e.g., Reed-Solomon codes, should be used to achieve better performance; otherwise, a simple XOR FEC scheme can be used for limited improvement.

Chapter 5

Securing the Reprogramming Process

Reprogramming is performed via wireless radio, which is a broadcast medium, and is vulnerable to packet injection or corruption attacks. Moreover, the current reprogramming protocols [23, 30, 33, 47, 60] are epidemic in nature. Once a false or viral code image is installed on one sensor, it could rapidly infect the entire network, and thus, lead to catastrophic damage. For these reasons, it is important that sensor nodes be able to verify that the code image is from a trusted source.

In this chapter, we focus on the security of protocols such as MNP discussed in Chapter 2. Specifically, we focus on authentication. Our goal is to provide a way that sensor nodes can verify program authenticity and integrity. The major challenge of this problem in the context of sensor networks is that the amount of available RAM is significantly small (e.g., Mica2 sensors [7] have only 4KB RAM). Therefore, if large amount of data needs to be sent to the sensor nodes then this data must be stored on EEPROM, which is much larger in size. One of the important problems with this however is that EEPROM writes are expensive in terms of energy (e.g., writing a 16-byte block to an EEPROM is approximately four times more expensive than transmitting a message (cf. Section 1.2)). Moreover, each EEPROM location can be successfully written only a finite number of times (typically about 10,000 operations [13]). In other words, after a certain number of writes, the EEPROM location cannot be changed subsequently. Thus, an adversary can launch a denial of service (DOS) attack by sending garbage data to a sensor. Even if the sensor later finds out that the data is invalid, it would have had spent significant energy in saving the data to EEPROM. This suggests that in the presence of denial of service attacks, data must be authenticated *before* it is written to EEPROM. We define this problem as the problem of bulk data dissemination, with reprogramming as one special case.

Although our protocol is designed for reprogramming, it can be applied in other bulk data dissemination scenarios as well. Specifically, we consider the problem of bulk data dissemination in three scenarios. In the first scenario, bulk data dissemination needs to be solved for reprogramming a sensor network where the base station sends the new program to the sensors. This program size is typically tens of kilobytes in typical applications. In the second scenario, the data is moderate in size. Such a scenario may occur if the base station is collecting statistics about the network performance. The base station may then send a summary of such data along with new commands, reconfigured values of different parameters, code for revised functionality, etc., to all the sensors in the network. Another example is difference-based reprogramming, in which the base station only sends the differences of the new reprogram, rather than the entire binary image, to the sensors. It is expected that the size of such data would typically be smaller (1-4 KB) compared to the case of reprogramming where the old program is completely replaced by the new program. However, it can still be large enough that storing it entirely in memory may not be feasible, especially due to the fact that typical applications use static memory allocation for main memory. The third scenario occurs in cases where the network is divided into a collection of (possibly overlapping) clusters. In such a scenario, the cluster leader would need to communicate data to all sensors in its cluster. This scenario differs from the second scenario in that the communication within the cluster is expected to be single hop in the third scenario whereas the communication is expected to be multihop in the second scenario. A variation of the third scenario also occurs when reprogramming needs to be done in a laboratory environment where all sensors are close to the base station (i.e., within distance 1). Even in such a scenario, security is needed to ensure that two users trying to reprogram their respective sensors do not (accidentally or otherwise) interfere with each other.

The organization of this chapter is as follows. In Section 5.1, we describe the threat model and security requirements of the secure bulk data dissemination problem. In Section 5.2, we present the basic hash chain scheme for authenticating a data stream. In Section 5.3, we discuss the cost of signing data in sensor networks. In Section 5.4, we introduce the secret instantiation algorithm from [18,31] that we use in our protocol to provide authentication.

According to their capacities, the adversaries can be categorized into two groups: mote-class adversaries and laptop-class adversaries. Mote-class adversaries have limited energy, and cannot launch extensive denial of service attacks. A laptop-class adversary can launch denial of service attacks by injecting a large number of garbage packets to the network. In Section 5.5, we present our authentication protocol for the case where only mote-class adversaries exist. We use MNP as an example, and show the performance. Then, in Section 5.6, we improve this protocol so that it also deals with denial of service attacks from laptop-class adversaries. We illustrate the applicability of our approach in the three scenarios considered above and evaluate the performance. In Section 5.7, we propose additional techniques to improve the performance for the first scenario (i.e., reprogramming a sensor network), and evaluate the performance enhancement by applying these techniques. In Section 5.8, we discuss issues on key distribution and updates. We summarize this chapter in Section 5.9.

5.1 Threat Model and Security Requirements

We consider an adversary as one who tries to inject its own code into sensor nodes or launch denial of service attacks that aim to exhaust sensors' battery power. It can eavesdrop on any communication in the network. It is able to compromise a sensor node, and acquire all information inside it. It can also inject, change, delete packets. However, an adversary cannot compromise the base station, which is securely protected.

We focus on authentication only, i.e., we assume that confidentiality is not required, i.e., the data being transmitted are public and can be acquired by the adversary. Hence, the data are sent in plain text along with appropriate authentication.

The goals of the proposed protocol are as follows:

1. Authenticity. Each sensor must be able to verify that data are from a trusted source and have not been changed during transit. We consider the base station as a trusted source, and is protected against compromise.

2. Node-compromise resilience. It must not be possible that compromising a single sensor node will cause the other parts of the network insecure.

3. Low cost. The security scheme should be efficient in terms of computation, communication, memory usage, and energy consumption. Moreover, it should not add long delay to the data dissemination process.

Moreover, in the case where laptop-class adversaries exist, a sensor should verify the authenticity and integrity of a received packet before writing it to flash. This is to reduce the energy cost of receiving fake packets from an adversary in a denial of service attack.

5.2 Scheme for authenticating a data stream

One way to authenticate a data stream is to use the approach from [17] for signing digital streams (as done in [15]). Assume that the entire program has N ($N \geq 1$) segments. Each segment contains K packets (possibly with the exception of the last segment). We represent the j^{th} data packet of the i^{th} segment as $P(i,j)$, $i = 1..N$, $j = 1..K$ (we also refer to it as packet j for simplicity, as long as it does not cause confusion). Hash of $P(i,K)$ is computed and attached to $P(i,K-1)$. Then, hash is computed on this modified packet (i.e., packet $P(i,K-1)$ and hash of $P(i,K)$) and attached to $P(i,K-2)$. This process is then repeated until we obtain $P(i,1)$ and the hash of the modified $P(i,2)$ (that contains $P(i,2)$ and hash of *modified* $P(i,3)$). In this way, we construct one hash chain per segment. Finally, the hash of the modified first packet is then *signed.* We call this approach *the basic hash chain scheme.*

In Figure 5.1, we show the basic hash chain scheme for segment i. The hash of packet $P(i,j)$ is denoted as $H(i,j)$. A data packet $P(i,j)$ has two parts, the data part and a hash of the next packet (not shown in Figure 5.1 for clarity). In Figure 5.1, an arrow pointing from packet j to packet i indicates that packet i contains the hash of packet j. If $P(i,j)$ is the last packet of segment i ($1 \leq i < N$), then the hash is 0. Hence, a data packet in a basic hash chain can be represented as in Figure 5.2.

Note that the hash is computed over the entire packet, not just the data part. The base station signs the hash of the first packet in the segment, which is the head of the hash chain, using all the secrets. We denote the signatures of segment i as $sign(H(i,1))$ in Figure 5.1.

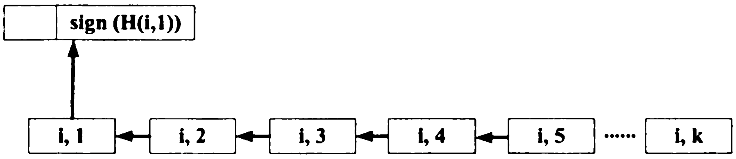


Figure 5.1: The basic hash chain (segment i).

```
if $P(i,j)$ is the last packet of segment $i$
    $P(i,j) = data(i,j)||0, i = 1..N, j = K$
else
    $P(i,j) = data(i,j)||H(i,j+1), i = 1..N, j = 1..(K-1)$
endif
```

Figure 5.2: Representation of a data packet $P(i,j)$ in the basic hash chain.

With the basic hash chain approach, when a node receives the first packet, it can use the signature to authenticate it. Additionally, it obtains the hash value for the (modified) second packet. Thus, when it receives the second packet, it can use the hash value to verify it, and so on. Note that a sensor can verify a data packet $P(i,j)$ if and only if it has received and verified all the packets in the hash chain proceeding $P(i,j)$. It implies that the data packets have to be *verified* in order. This is inefficient in the events of packet loss/delay. For example, if all the packets in segment i have been received except packet 2, none of the packets after packet 2 in the chain can be verified. In the case where denial of service attacks exist, such packets cannot be stored in EEPROM. They have to be thrown away if there is not enough memory to cache them.

As we can see, in the basic hash chain scheme, a single packet loss/delay can lead to erasure of many valid data packets. This leads to significant energy waste. Our simulation results (as well as the results from [14, 15]) show that if we require that the data packets be received and stored in order, data dissemination process will be delayed significantly (e.g., the completion time increases by 6 or 7 times if we use the default segment size 128 packets/segment in MNP). Correspondingly, the energy consumption also increases by a large amount.

5.3 Cost of Signatures In Sensor Networks

In the basic hash chain scheme, the important issue is the approach used for signing the hash of the first packet. If the *cost* of this signature is reduced then it would assist in reducing the overall cost. The cost of the signature is especially important in the second and third scenarios described at the beginning of this chapter, where the size of the data is comparatively smaller.

Existing approaches [14, 15, 34] use public keys for authenticating the data used in reprogramming. The base station signs the hash of the first packet using its private key and each sensor decrypts it using the corresponding public key. However, creating and verifying the asymmetric digital signatures have very high computation overhead. Moreover, since the cryptographic operations are overlapped with the radio operations; if the encryption/decryption operations are not fast enough, we may encounter problems if the radio packets needed by the sensors are no longer available. Although recent work has shown that RSA and elliptic curve cryptography (ECC) are feasible on Mica/TelosB motes [20, 45, 64], they should still be avoided or used sparingly.

With this motivation, we focus on use of symmetric keys, which need much less energy/memory/computation resources, and hence, are expected to be more appropriate for resource constrained sensor nodes. For example, if we use Skipjack (or RC5) on Mica2 motes, the execution time for encrypting/decrypting an 8-byte block is only 0.38ms (or 0.26ms in the case of RC5), which is less than the time for sending one byte data over radio [25]. This approach also has the potential to allow intermediate packets to be signed so that even if a sensor misses some packet, it can authenticate and store packets received after the subsequent signature. Inserting such intermediate signatures is feasible since the cost of computing these signatures is small. A simple approach is to use a single network-wide key shared by the base station and all the sensors [25]. The problem with this approach is that a sensor cannot verify if the data received is from the base station or another sensor in the network. Therefore, we must use symmetric keys in such a way that the nodes can verify that the data was indeed sent by the base station.

We propose a symmetric key based protocol that authenticates the bulk data dissemination process in sensor networks. We use a secret instantiation algorithm from [18, 31] that we will describe in Section 5.4 to provide authentication. The algorithm requires only $O(\log n)$ keys to be maintained at each sensor. Thus, in our protocol, only a very small number of keys are maintained at every sensor.

5.4 Protocol For Signing The Hash of The First Packet

In this section, we introduce the secret instantiation algorithm [18, 31] that we use in our symmetric key based authentication protocol. This algorithm requires only $O(\log n)$ keys to be maintained at each sensor.

The base station has a collection of secrets. Initially, each sensor receives some subset of this collection. Whenever the base station sends a message, it separately signs it using all the secrets in its collection. Thus, message transmission is associated with a collection of signatures, one for each secret that the base station has. To sign message m, with secret s, the base station can use algorithms such as MD5. (Additionally, if the length of the signature needs to be small, then only a small part of this signature (e.g., last few bytes) may be used.) Whenever a sensor receives this communication, it verifies the signatures based on the collection of secrets it has. Of course, a sensor will only be able to verify a subset of the signatures, as it does not have all the secrets. It is required that if all these signature verifications are successful, the sensor can assume that the communication is truly from the base station (and not from an outsider or anther sensor pretending to be the base station).

To implement this algorithm, a 2-dimensional array of secrets with r rows (numbered $0..r-1$) and $\log_r n$ (numbered $1..\log_r n$) columns (where $2 \leq r \leq n$ and n is the number of sensors) is maintained. The base station knows all these secrets. Each sensor is assigned a unique ID that is a number with radix r. Observe that the ID is of length $log_r n$. (Leading 0s are added if necessary.) This ID identifies the secrets that a sensor should get. Specifically, if the first digit (most significant) of the ID is x then the sensor gets x^{th} secret in the first column. If the second digit of the ID is y then the sensor gets y^{th} secret in the second column, and so on.

To illustrate the algorithm, we show an example in Figure 5.3. Let the number of nodes be 16 and let r be 2. Then the base station contains 8 (i.e., $2\log_2 16$) secrets with 2 rows and 4 columns. Each sensor has 4 (i.e., $\log_2 16$) secrets. The set of secrets a sensor has are decided by its unique ID. For example, if a sensor's ID is 0011, then it has the secrets on the first row in the first two columns and the secrets on the second row in the next two columns.

Theorem 1. If sensor j receives a message and it verifies all the signatures based on the secrets it knows then that message must be sent by the base station.

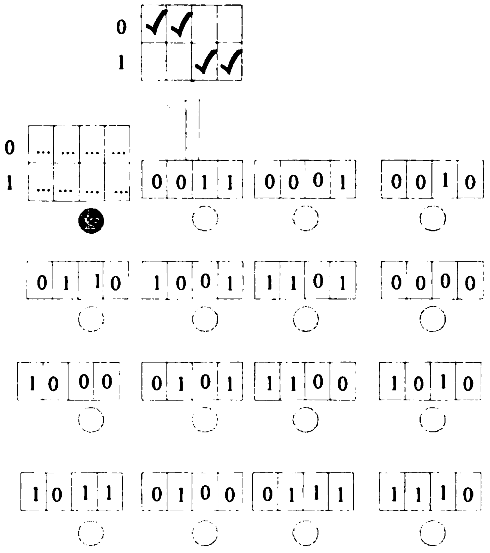


Figure 5.3: Secret instantiation: an example.

Proof: Each sensor has a unique ID that is of length $log_r n$, thus it is associated with a unique combination of $log_r n$ secrets. Only the base station contains all the secrets. Therefore, no other sensor, except the base station, has all the secrets that sensor j has. Hence, if j verifies $log_r n$ signatures, it is assured that the message originated at the base station. [18] □

Collusion. In the secret instantiation algorithm, compromising a single sensor node will not compromise the entire network. This is due to the facts that each sensor has only a subset of the secrets, and if an adversary attempts to pretend to be the base station, it needs to get all the secrets. However, colluding sensors may be able to obtain all the keys and, thereby, pretend to be the base station. By choosing an appropriate value for r, this key distribution provides a tradeoff between level of collusion resistance and number of keys at the base station.

In our simulations, for simplicity, we used the base $r = 2$ thereby choosing the least number of secrets at the base station. Hence, in a 10x10 network (100 sensors), the base station maintains 14 ($2log_2 100$) secrets and each sensor maintains 7 ($log_2 100$) secrets. In this case, collusion of 2 users with complementary IDs (e.g., a sensor with ID 1010 and a sensor with ID 0101) can allow them to pretend to be the base station. If higher collusion resistance is desired, the designer can choose a higher base. For example, if $r = 10$ is used for a 10x10 network, then the number of secrets maintained at the base station increases to 20 (as compared to 14 when $r = 2$). Since these secrets are used only a few times during data dissemination, it will not affect the performance (time/energy) significantly.

On the other hand, with increased value for r, not only the collusion resistance increases, but also the number of secrets maintained by each sensor ($log_r n$) decreases. Thus, providing higher level of collusion resistance does not adversely affect the sensors. For $r = \sqrt{n}$, the algorithm corresponds [31] to the grid algorithm in [32]. For $r = n$, the algorithm corresponds to the case where each sensor maintains a unique secret that is known only to that sensor and the base station. In this case, collusion between sensors does not allow them to pretend to be the base station.

Computation cost of signing/verifying the signatures and computing the hashes. The hash chain and signatures are computed only once at the base station. The sensors simply use/forward the signatures received from the base station. Moreover, since we use symmetric keys for signatures, the overhead is much lower than (about 0.0005 times) the cost of using the asymmetric keys. While hash computation is performed for every packet, it is very efficient (less than 10ms per packet). Hence, hash computation does not significantly increase the computation cost either.

5.5 Authentication Protocol For Mote-Class Adversaries

In this section, we focus on the case where only mote-class adversaries exist. Examples of such adversaries are likely to occur in sensor network testbeds. Such testbeds are expected to be typically physically secure so that attacks from a laptop class adversary are prevented/mitigated. However, since the testbed relinquishes control of sensors to users for their experiments, one experiment can be affected by another concurrent experiment. In this case, a potential adversary is in mote-class, i.e., its computation and communication capability as well as battery power is similar to the sensors in the network. Our algorithm provides protection from such interference/attacks with a low overhead. We describe the protocol in Section 5.5.1, and present the evaluation results in Section 5.5.2.

5.5.1 Protocol Description

We use the basic hash chain approach described in Section 5.2 to construct one hash chain per segment. The hash of the first packet of each segment is signed using the secret instantiation algorithm [18,31] that we described in Section 5.4. We present our authentication protocol in the context of MNP (cf. Chapter 2). In MNP, a sender broadcasts a "StartDownload" message several times, then starts transmitting the requested data packets in the segment. We add the signature and the hash $H(1,1)$ in the "StartDownload" message. If the signature plus hash is too long, and does not fit in one message, we can use multiple messages for "StartDownload". In this case, the receiver needs to receive all of them in order to get the entire signature.

As we discussed in Section 5.2, in the basic hash chain approach, packets are verified in order. However, packets do not arrive in the same order as they are sent due to packet losses and/or delay. If we require that the data packets have to be *received/stored* in sequential order, we have to throw away a large portion of the data packets that have been received, which incurs significant energy waste. On the other hand, if out of order delivery is allowed then an adversary can mount a denial of service attack by sending a large number of garbage packets, as storing the packets (to EEPROM) requires significant energy. However, since a mote class adversary has limited power and message transmission requires significant energy, the adversary cannot launch such attack for a large duration. Therefore, in our design, we allow the packets that arrive out of order (within one segment) to be received and stored immediately.

In our protocol, each sensor maintains a *received* vector (corresponds to *MissingVector* in MNP), which is a bitmap of the current segment, indicating which packets have been received. All the bits in the *received* vector are initialized to 0. Once a sensor receives a packet for the first time, it stores the packet (the data part) in EEPROM, and sets the corresponding bit in *received* to 1. The hash value part of the packet needs be buffered in memory so that it can be retrieved fast for verification. Moreover, each sensor contains a variable, *verifiedPackets*, which is the number of packets the sensor has received and verified.

When a sensor node receives a packet $P(i, j)$, if the *previous* packet has been verified, i.e., $j =$ *verifiedPackets*+1, then the node can verify packet $P(i, j)$ using the saved (and verified) $H(i, j)$ from the previous packet. If the verification is successful, the node will increase *verifiedPackets* by 1, and continue verifying the *next* received (but not verified) packet. (When we talk about *previous* or *next* packet, we refer to the previous or next node on the hash chain.) This verification process continues until we reach a missing packet. On the other hand, if the packet verification fails, the packet is thrown away (i.e., *received*(j) is set to 0), and the verification process stops.

The operation a sensor performs when it receives a data packet $P(i, j)$ (the j^{th} packet in the i^{th} segment) is shown in Figure 5.4.

```
when a data packet P(i,j) arrives
    if P(i,j) is received the first time, i.e., received(j) == 0
        store P(i,j): store data part in EEPROM and the hash value for the next
            packet in memory, set received(j) to 1
        while ((received(j) == 1) and (j == verifiedPackets + 1))
            Compute H(i,j)
            if computed H(i,j) == saved H(i,j) from the previous packet
                verifiedPackets++, j++
            else // the packet cannot be verified, hence is thrown away
                set received(j) to 0, break while loop
            endif
        endwhile
    endif
```

Figure 5.4: Operation a node performs when it receives a data packet P(i,j)

5.5.2 Evaluation

In this section, we evaluate the performance of our secure reprogramming protocol. We integrate our protocol with MNP as described in Section 5.5.1, and refer to the integrated protocol as *Secure*$_M$*MNP* (i.e., *SecureMNP* for Mote-class adversaries).

We simulate *Secure$_M$MNP* using TOSSIM [37]. We evaluate the performance in terms of memory requirement, delay, energy consumption, and communication cost. When we compute the energy consumption, we use Equation 2.1 from Section 2.2.2 in Chapter 2. Note that in this equation, we only consider the energy cost of communication, idle listening and EEPROM read/write. The energy cost of computing hashes and signatures is not included in Equation 2.1, but is discussed at the end of Section 5.4.

In our simulation, each segment has 128 data packets. The simulations are performed in a grid topology. The inter-node distance is 10 feet. Due to the fact that the execution time of each simulation is of order of tens of hours, we do not provide confidence intervals.

The default payload size of each packet in MNP is 23 bytes. Each packet carries a 4-byte hash, which is the hash value of the next packet. Hence, excluding the hash value, the *effective* data payload is 19 bytes. Therefore, in every data packet, 4 out of 23 bytes of the payload is consumed in authentication. Therefore, in order to transmit a program of certain size, more data packets need to be received at every sensor.

We consider a 20x20 network, i.e., the number of sensors in the network is 400. We set r to be 2. In this case, the base station contains 18 (i. e., $2\log_2 400$) secrets, and each sensor has 9 secrets. As we described in Section 5.5.1, the base station signs $H(1,1)$ using all the secrets, and attaches all the signatures and $H(1,1)$ in "StartDownload" messages. If sensors only verify the last bytes of the signatures, we only need 18 bytes for the signatures, and the signatures fit in a single packet. If two bytes of each signature are used, then the length of the 18 signatures is 36 bytes. In this case, two packets for "StartDownload" messages need to be sent.

Memory requirement. Our authentication protocol has memory cost in the following ways. First, $\log_r n$ secrets are maintained at each sensor. When r increases, the number of secrets maintained at the sensor decreases. In a 20x20 network, the number of secrets at each sensor is no more than 9. In a 10x10 network, the number of secrets maintained at each sensor is at most 7. Second, the signatures from the base station for each segment need to be stored either in memory or in flash. Third, the signing/verification process consumes some amount of memory. This amount of memory is much lower compared to the asymmetric key based approaches. Fourth, the algorithm uses a variable *verifiedPackets* to keep track of how many packets have been verified/authenticated. Fifth, since we allow packets to be received and stored out of order, we need to store all the hash values for the packets in the current segment. As we assume a 4-byte hash value is used and each segment contains 128 packets, the space that is used to store hash values is 512 bytes. The hash values can be stored in memory if reprogramming speed is important.

Delay and energy consumption. We assume that the last two bytes of each signature are used, then the collection of the signatures from the base station are contained in two "StartDownload" messages: "StartDownload1" and "StartDownload2". In order to get the entire set of signatures, each node needs to receive both messages. As we allow packets to be saved before verified, and hash values can be computed very fast, authentication process does not affect reprogramming time significantly. Given a certain number of data packets to be sent, the reprogramming time remains almost the same no matter whether the security protocol is used. The major overhead is the hash values that are carried in data payload. As we have pointed out earlier, 4 out of 23 bytes data payload in a data packet are used for authentication. The size of code image that is sent by a segment in MNP is 2.94KB (128 × 23 bytes). By contrast, the size of the code image that is sent by a segment in $\mathit{Secure}_M \mathit{MNP}$ is 2.43KB (128 × 19 bytes). In Figure 5.5, we show that given a certain amount of code image to be sent, the reprogramming time required by $\mathit{Secure}_M \mathit{MNP}$ is only a little (1-16%) higher than that required by MNP. The energy consumption of $\mathit{Secure}_M \mathit{MNP}$ is 7-54% higher than that of MNP.

Communication cost. Similarly, the communication cost required by

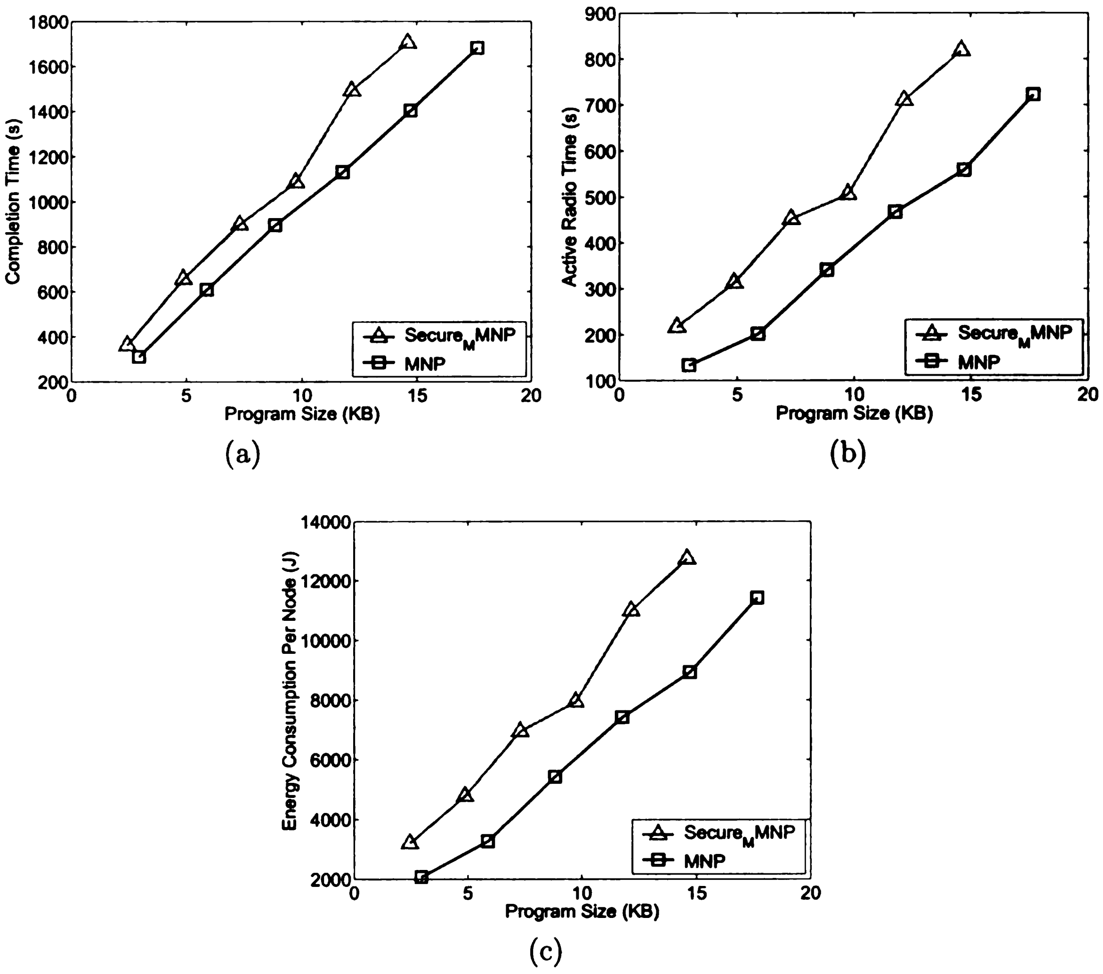
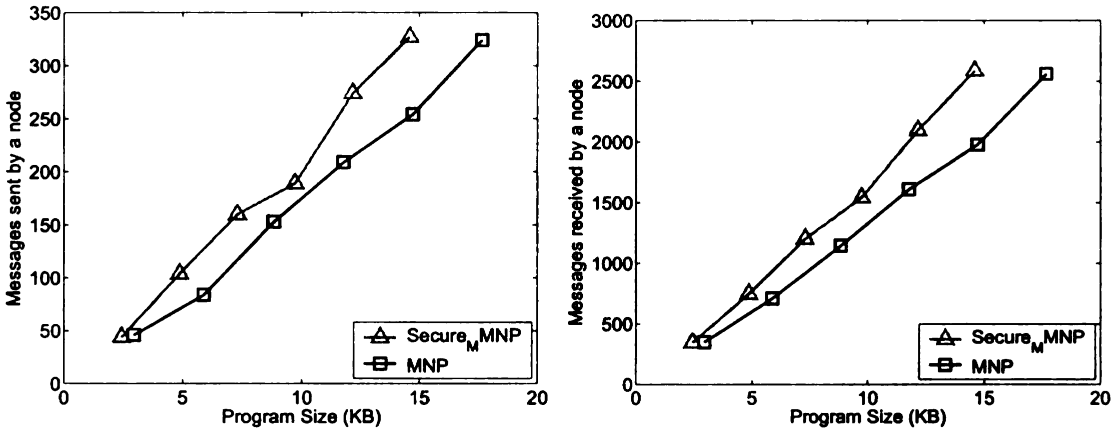


Figure 5.5: Comparing MNP and *Secure$_M$MNP*: delay and energy consumption of authentication under different program sizes. (a) completion time vs. program size (b) active radio time vs. program size (c) energy consumption per node (not including the authentication cost) vs. program size.

Secure$_M$MNP is a little higher than that is required by MNP due to fact that the hash values that are attached with every packet. In Figure 5.6, we show that for a given program size to be distributed, the message transmission and reception of *Secure$_M$MNP* is about 20% higher than that of MNP.



(a)

(b)

Figure 5.6: Comparing MNP and *Secure$_M$MNP*: communication cost of authentication under different program sizes. (a) message transmission per node vs. program size (b) message reception per node vs. program size.

5.6 Authentication Protocol For Laptop-Class Adversaries

In this section, we consider the case where laptop-class adversaries exist. To mitigate the denial of service attacks from a laptop-class adversary, we require that the received data packets must be authenticated *before* stored to EEPROM. We show the effectiveness of our approach in the three scenarios discussed at the beginning of this chapter. First, in Section 5.6.1, we discuss the third scenario where the sensors are within a single hop of the source. Subsequently, in Section 5.6.2, we discuss the second scenario where a moderate amount of data (1-4KB) is sent using a *fine-grained pipelining* (i.e., packet-level pipelining) protocol (e.g., Infuse [30]). In both scenarios, we show that the time to transmit moderate amount of data is less than the time for transmitting even **a single packet** with public keys. In Section 5.6.3, we discuss the first scenario where large amount of data is sent using a *coarse-grained pipelining* (i.e., segment-level pipelining) protocol. We focus on the reprogramming protocol MNP (cf. Chapter 2) and show that security can be added to it using our approach. Furthermore, we illustrate how adding redundancy to the transmitted data can further reduce the cost of adding security.

5.6.1 Secure Single-hop Dissemination

In this section, we focus on the scenario where the source node is disseminating a moderate amount of data to all the receivers within single hop. This typically happens in a small/indoor network, or within a cluster in a large network. The source node can be the base station or a cluster head depending on how the protocol is used. Any single-hop/multi-hop reprogramming protocol (e.g., [12, 23, 30, 33, 47]) can be used for dissemination in this scenario. We use a simple CSMA-based protocol (similar to that in [12]), which is described as follows.

The base station computes hash for each data packet. The hashes are put into one or a few packets (called hash packets). The base station then computes hashes for these hash packets, and accommodates them into one or a few higher-level hash packets. In this way, the hashes of packets are organized into a hash tree (this uses the approach in [14]). The base station signs the root of the hash tree using all the secrets it has (based on our symmetric key algorithm). The base station sends the signatures at first, followed by the hash tree, from higher level or lower level. After the hash tree has been sent, it sends the data packets. Because the hashes are sent before the data packets, when a sensor receives a data packet, it can authenticate this packet immediately using the hash value. After the base station has transmitted all the data packets, it will send query several times. If the base station receives requests from the receivers, it will retransmit the requested packets. This continues until all the receivers have received all the hashes and data packets. We call this approach *secure single-hop dissemination with hash tree.*

Alternatively, we can also use the basic hash chain approach, as described in

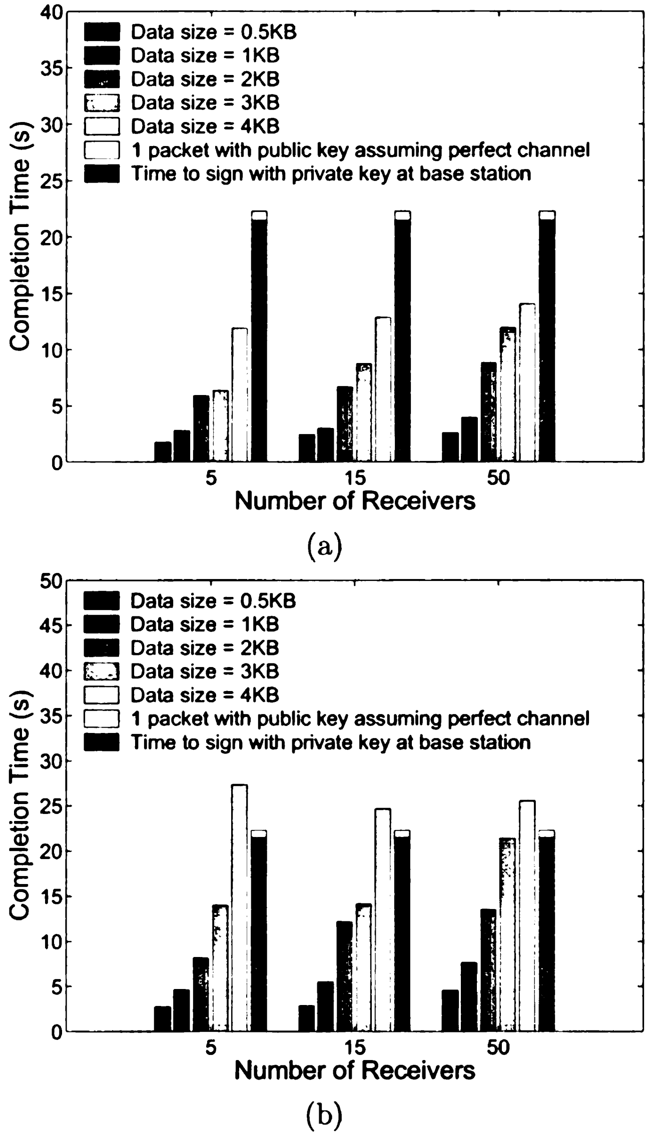
Section 5.2. The base station sends the signatures of the hash of the first packet, the first packet contains the hash of the second packet, and so on. We call it *secure single-hop dissemination with hash chain.*

We run this simple protocol (with hash tree and hash chain variations) on a special purpose simulator. The base station sends the signatures and the hash packets twice for robustness. The payload size of a data packet is 70 bytes, including 6 bytes header and 64 bytes data. A hash is 4 bytes long. Hence, each packet can carry 16 hashes. The signatures fit in one packet.

The transmission interval of a data packet is 45ms. When a receiver receives a query packet, it will wait for a short random duration before sending a request. We vary the data size from 0.5KB to 4KB, and simulated the two approaches at different receiver set sizes. Specifically, the number of receivers is set to 5, 15, 50. Packet loss rate is 5%. We repeat the simulations three times, and use the average.

In Figure 5.7, we show the completion time of this secure single hop protocol at different data sizes and different receiver set sizes. For comparison, we also show the time required to send **a single packet** through the network using public key scheme. According to [64], the time required for the base station to sign a packet with its private key is 21.5s. And, the time required to verify the packet at each sensor is 0.79s. Thus, the time to propagate one packet to a single hop network is at least 22.29s (21.5s+0.79s). From Figure 5.7, we can see that using our symmetric key based authentication with hash tree approach (respectively, with hash chain approach), the time to send 4KB data through the network (including request and retransmission time) is only 53-63% (respectively, is close to) the time to transmit **a single packet** to the network using public key based authentication. This shows that using symmetric key based authentication can significantly reduce the time (and energy) cost compared to the public key based authentication.

Memory requirement. The memory requirement of the protocol is listed as



(a)

(b)

Figure 5.7: Completion time in a single hop network. (a) hash tree approach (b) hash chain approach. (For approaches with public key schemes, only signing/verification cost is considered. Communication cost, although possibly significant when the size of signature is large, is not considered. But for our protocol, both signing/verication cost and communication cost are considered.)

follows. The amount of memory required for caching the secrets and the signatures, and that required during signing/verification process are the same as what we have discussed for secure reprogramming for mote-class adversaries (cf. Section 5.5.2). In addition, in the hash tree approach, each sensor caches all the received hashes in RAM (and use them to verify the data packets later). If the length of the data stream is 4KB (64 packets), the amount of memory for storing the hashes is 256 bytes. If the hash chain approach is used, only the hash contained in the last packet that has been authenticated needs to be cached in RAM. In this case, the RAM requirement for hash cache is only 4 bytes.

5.6.2 Secure Multihop Dissemination with Fine-Grained Pipelining

In this section, we focus on the scenario where the base station disseminates a moderate size data stream (of size 0.5-3KB) across the network in a *fine-grained pipelining* fashion. Such fine-grained pipelining service can be achieved using any TDMA based data dissemination protocol (e.g., Infuse [30], Sprinkler [47]). To illustrate secure dissemination in a multihop network with fine-grained pipelining, we use Infuse [30] to disseminate the data across the network. We note that the evaluation results presented in this section are applicable to other data dissemination protocols that use fine-grained pipelining.

Overview of Infuse. Infuse is a reliable TDMA based data dissemination protocol. Since we consider grid based networks in this dissertation, we present an overview of Infuse for such a network. The basic idea of Infuse is to assign time slots in such a way that no two nodes within distance two (in the grid) of each other should get the same slot. This ensures that whenever any node transmits data, nodes within distance 1 can always receive that message without collision. For the case where the communication range of a node exceeds the distance with the closest neighbor, nodes can be assigned different frequencies to prevent collision. Note that Mica motes can provide multiple channels (e.g., 54 channels in the 902-928 MHz frequency band) on which they can transmit [9]. However, they can listen to only one frequency at a time. To illustrate this, consider a line network a-b-c-d-e. Suppose that sensors can communicate/interfere with nodes upto distance 2 then, a and d should transmit on different frequencies. Depending on the communication/interference range of the sensors, the number of frequency channels required varies. The issue of number of frequency channels required is outside the scope of this dissertation. Although TDMA ensures collision freedom, messages can still be lost due to random channel errors. To deal with this, Infuse uses sliding window based recovery mechanism and *implicit acknowledgments* (by listening to the transmissions of the neighbors).

Evaluation with Infuse. We authenticate the data stream disseminated with Infuse using the basic hash chain approach discussed in Section 5.2. We call this protocol *SecureInfuse*. We simulate the protocol in Prowler [59], a probabilistic wireless network simulator for Mica motes. The goal of the simulations is to illustrate that the completion time with *SecureInfuse* is significantly less than that with public key scheme. We disseminate data of sizes 0.5-3KB across 10x1, 5x5, and 10x10 networks. The payload size of the messages transmitted by Infuse is 16 bytes. In our simulations, we use 4 packets as the sliding window size (for dealing with random channel errors). Since random channel errors can cause the link reliability to go down, we choose a conservative estimate of 95% link reliability in our simulations.

Figure 5.8 shows the completion time of *SecureInfuse*. As observed from the figure, the time to disseminate data with *SecureInfuse* is significantly less than the time to propagate one packet across the network using public key mechanism. Based on [64], with public key scheme, the analytical result on time required to sign and verify a packet across a 10x1 network (respectively, 5x5 and 10x10 networks) is 28.61s (respectively, 27.82s and 35.72s). On the other hand, with *SecureInfuse*, the time to disseminate the data stream of size 3KB is approximately the same as the time required with the public key scheme for **a single packet**. Thus, the time to securely disseminate a moderate size data across a multihop network is significantly less with the use of symmetric keys. (Note that due to multiple paths in a 10x10 network, a sensor may recover a lost packet from a different neighbor. For this reason, the transmission time is lower in a 10x10 network than in a 10x1 network. For details of such behavior, we refer the reader to [30], as this issue is not central to the topic of this dissertation.)

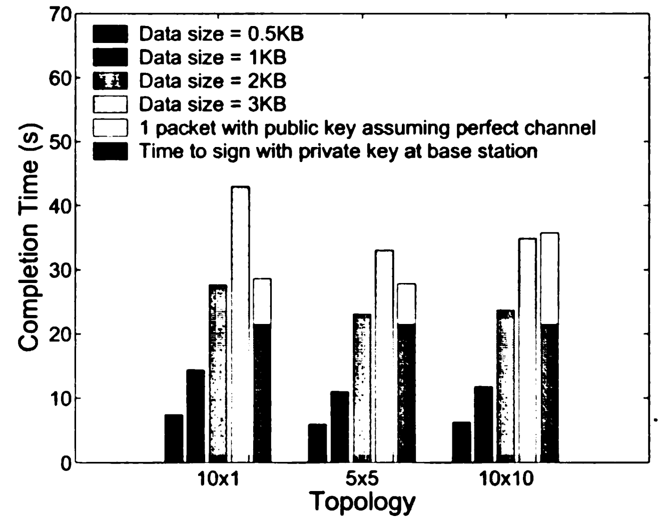


Figure 5.8: Completion time with window size = 4. (For approaches with public key schemes, only signing/verification cost is considered. Communication cost, although possibly significant when the size of signature is large, is not considered. But for our protocol, both signing/verication cost and communication cost are considered.)

Memory requirement. Similar to the discussion in Section 5.5.2, *Secure-Infuse* also maintains $\log_r n$ secrets at each sensor. Additionally, since *SecureInfuse* uses the basic hash chain approach, only the hash contained in the last authenticated packet needs to be kept in RAM. Finally, Infuse maintains a sliding window of 4 packets (=64 bytes, as the packet size in Infuse is 16 bytes) to deal with the problem of random message losses. However, this requirement is inherent to Infuse irrespective of whether authentication is enabled or not.

5.6.3 Secure Multihop Dissemination with Coarse-Grained Pipelining

Coarse-grained pipelining are implemented in protocols such as MNP (cf. Chapter 2) and Deluge [23]. In these protocols, the data stream is divided into segments and pipelining is provided at the segment level. Security of Deluge is considered in [15] and uses the scheme in [17] that is described in Section 5.2. Using our approach would be identical except that our approach reduces the cost of signing and verifying the hash of the first packet. Thus, using our approach would reduce the cost of data propagation by approximately 22 seconds than that in [15]. Likewise, if hash tree approach is used with symmetric keys then the data propagation time would improve by approximately 22 seconds over that in [14]. For this reason as well as the fact that the data propagation time in [14, 15] is in hundreds of seconds, we do not provide detailed simulation results for this scenario. Instead, in the next section, we present additional mechanisms that would reduce the cost of secure data dissemination in coarse-grained pipelining.

5.7 Performance Enhancement of Authentication Protocol For Laptop-Class Adversaries

In this section, we propose three techniques to improve the performance of our authentication protocol when it is used to disseminate a large amount of data in a multihop network with coarse-grained pipelining. We do a case study on MNP (cf. Chapter 2). We note that the design and simulation results we present in this section are applicable to other data dissemination protocols as well.

In Section 5.7.1, we describe how we use the keys and hashes to sign the data stream. Specifically, we propose the *double connected hash chains*, combined with symmetric key signatures, to authenticate the data stream. We also propose two other schemes to further improve the efficiency: creating a cache on the receiver side (Section 5.7.2) and using forward error correction (FEC) (Section 5.7.3). We evaluate the performance of our authentication protocol and show the effect of applying these schemes in Section 5.7.4.

5.7.1 Double Connected Hash Chain

In the basic hash chain scheme (cf. Section 5.2), the data packets have to be *verified* in order. Losing a single data packet can lead to all the succeeding packets to be thrown away. This leads to significant time/energy waste. To address this problem, we propose a double connected hash chain to enhance the inter-connection among the packets, as illustrated in Figure 5.9. Assume that the entire data stream has N ($N \geq 1$) segments and each segment contains K packets. We represent the j^{th} data packet of the i^{th} segment as $P(i, j)$, $i = 1..N$, $j = 1..K$. (We also refer to it as packet j for simplicity, as long as it does not cause confusion.) The hash of packet $P(i, j)$ is denoted as $H(i, j)$. As shown in Figure 5.9, a segment is further divided into *hash groups*. Each hash group contains m packets. m is an integer factor of K. A packet $P(i, j)$ contains a data part and two hashes: the hash of the next packet (with successive packet ID, e.g., $P(i, j + 1)$), and the hash of the corresponding packet in the next hash group (e.g., $P(i, j + m)$). In Figure 5.9, an arrow pointing from packet j to packet i indicates that packet i contains the hash of packet j. In Figure 5.10, we represent a packet $P(i, j)$ in a double connected hash chain. In this way, we construct multiple authentication paths for verifying a data packet. To illustrate how the double connected hash chain works, consider the scenario where a single packet (packet 2) is lost, while all other packets in the segment have been received. If we use the double connected hash chain, all the packets starting from packet m+1 can be authenticated because packet 1 contains the hash for packet m+1.

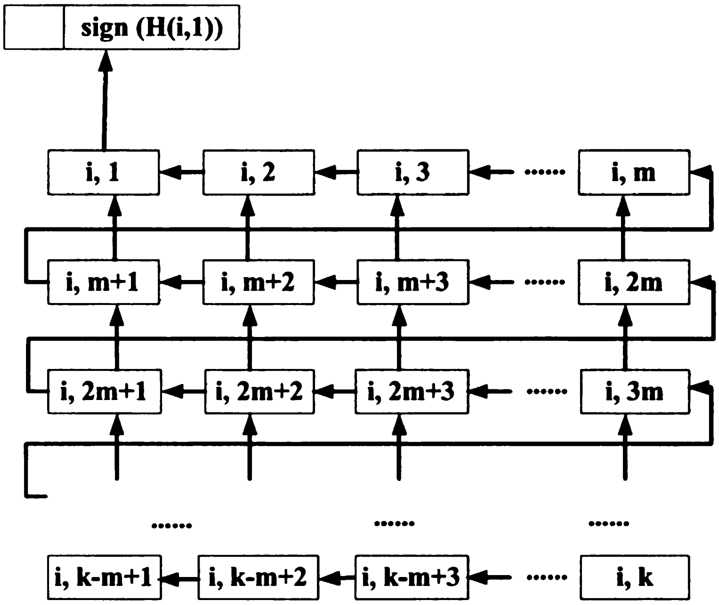


Figure 5.9: The double connected hash chain (segment i).

```
if $P(i,j)$ is the last packet of segment $i$
    $P(i,j) = data(i,j)||0||0,$
        $i = 1..N, j = K$
else if $P(i,j)$ is in the last hash group of segment $i$
    $P(i,j) = data(i,j)||H(i,j+1)||0,$
        $i = 1..N, j = (K-m+1)..(K-1)$
else
    $P(i,j) = data(i,j)||H(i,j+1)||H(i,j+m),$
        $i = 1..N, j = 1..(K-m)$
endif
```

Figure 5.10: Representation of a data packet $P(i,j)$ in the double connected hash chain.

5.7.2 Caching

Data cache and hash cache. We use two caches: a data cache, for storing the data packets, and a hash cache, for storing the hashes. Assuming a hash is 4 bytes long and the size of a segment is 64 packets, we only need 256 bytes to store all the hashes for the entire segment. On the other hand, storing data packets requires much more space. For example, if we set the packet size to 70 bytes, excluding 6 bytes for the header, storing one packet needs 64 bytes. In this case, caching the a 64-packet segment requires 4KB memory, which exceeds the memory capacity of some sensor platforms, such as Mica2 motes. To reduce the memory consumption, we divide a segment into *cache groups*. At any given time, each sensor caches data packets only from one cache group. The cache group that is currently kept in the data cache is called *the active cache group*. Note that when a sensor writes a data packet to its data cache, it writes the data part as well as the two hashes. By doing this, when it authenticates a packet, the hashes contained in the packet are authenticated at the same time. Only those hashes that have been authenticated are written to the hash cache.

When a receiver receives a data packet, if it needs the packet, it computes the cache group this packet belongs to. If the packet is in the active cache group, the sensor stores the packet to its assigned slot in the data cache. Otherwise, it changes the active cache group to the one that this data packet belongs to, and stores the received packet to the data cache. When a sensor changes its active cache group, if there are data packets in the data cache that are not yet authenticated, these packets are discarded. Moreover, when a sensor writes a packet to the data cache, it also checks if the hash cache contains the hash for this packet. If so, the packet is authenticated: the data part of the packet is written to EEPROM, and the hashes are written to the hash cache (if they are not already in the hash cache).

When a hash is written to the hash cache, the sensor checks the data cache to see if the newly added hash can be used to authenticate any data packet. If so, the packet is authenticated, and the two hashes contained in this packet can be added to the hash cache, which could in turn be used to authenticate more packets in the data cache. Hence, one reception of a data packet can possibly lead to several continuous writes to EEPROM. These writes to EEPROM must be queued and data are buffered when necessary so that the erasure of the data cache will not cause loss of data.

5.7.3 Forward Error Correction

MNP, as well as all other existing data dissemination protocols [23,30,47,60], uses automatic repeat request (ARQ) scheme to recover the lost packets. In ARQ schemes, a receiver detects its own losses, and informs the sender of the missing packets, either by sending requests or acknowledgments. The sender retransmits the packets that are requested by the receivers. In the current problem, a single lost packet may cause a sensor to discard other (valid but not yet authenticated) packets. Hence, in order to reduce packet loss, in our protocol, we use forward error correction (FEC).

FEC provides reliability by transmitting redundant packets in a proactive manner. Due to computational limits on sensors, we use the simple XOR FEC scheme. For simple XOR code, each transmission group has only one parity packet, which is the XOR or all the source packets in the group. XOR code is very simple to implement, and it can repair a single packet loss in a transmission group.

The data cache provides required memory space for encoding and decoding XOR parity packets, hence, there is no additional overhead on memory consumption for employing FEC. In our approach, a sender transmits a parity packet after transmitting t data packets (t is the size of the transmission group). The parity packet is XOR of the t data packets that proceeding it. We require that t be not larger than the size of the data cache.

When a receiver receives a parity packet, it checks if exactly one packet is missing in the transmission group that this parity packet belongs to. If so, it uses the parity packet to recover the packet that is missing. When a receiver tries to fix a missing packet by decoding a parity packet, if all the received packets in this transmission group are cached in the data cache (i.e., they are in the active cache group), decoding the parity packet can be conducted directly. In the case that some data packets were received in earlier transmissions and are not in the data cache, they must be read from EEPROM to the data cache. As reading a packet from EEPROM is an optimized operation with low cost, employing simple XOR code in our protocol does not incur much time/energy overhead.

5.7.4 Evaluation of Enhancement

We integrate our protocol with MNP (cf., Chapter 2), and refer to the integrated protocol as *Secure$_L$MNP* (i.e., *SecureMNP* for Laptop-class adversaries). We simulate *Secure$_L$MNP* using TOSSIM. In our simulation, each segment has 64 data packets. The size of a hash group m is 8. The simulations are performed in a grid topology. The base station is at the corner of the network. The inter-node distance is 10 feet. We consider a 10x10 network, i.e., the number of sensors in the network is 100. We set r to be 2. In this case, the base station contains 14 (i.e., $2\log_2 100$) secrets, and each sensor has 7 secrets. Due to the fact that the execution time of each simulation is of order of tens of hours, we do not provide confidence intervals.

We set the packet size to 70 bytes, among which, 6 bytes are used for the packet header (including source node ID, destination node ID, program ID, segment ID, packet type), the remaining 64 bytes are for the data and hashes. In *Secure$_L$MNP*, each data packet carries 2 hashes. Each hash is 4 bytes long. Hence, excluding the hashes, the *effective* data payload is 56 bytes. Therefore, in every data packet, 8 out of 64 bytes of the payload is consumed in authentication.

We first analyze the memory requirement of *Secure$_L$MNP* in Section 5.7.4. Then, we show the performance of *Secure$_L$MNP* in terms of completion time, active radio time, energy consumption, and communication overhead. The energy consumption is computed using Equation 2.1 from Section 2.2.2 in Chapter 2. As we mentioned in Section 5.5.2, Equation 2.1 does not include the energy cost on authentication (computing hashes and signatures), which has been discussed in Section 5.4. Finally, in Section 5.7.4, we investigate how our design decisions (i.e., double connected hash chain, caching and FEC techniques) affect the performance.

Memory Requirement

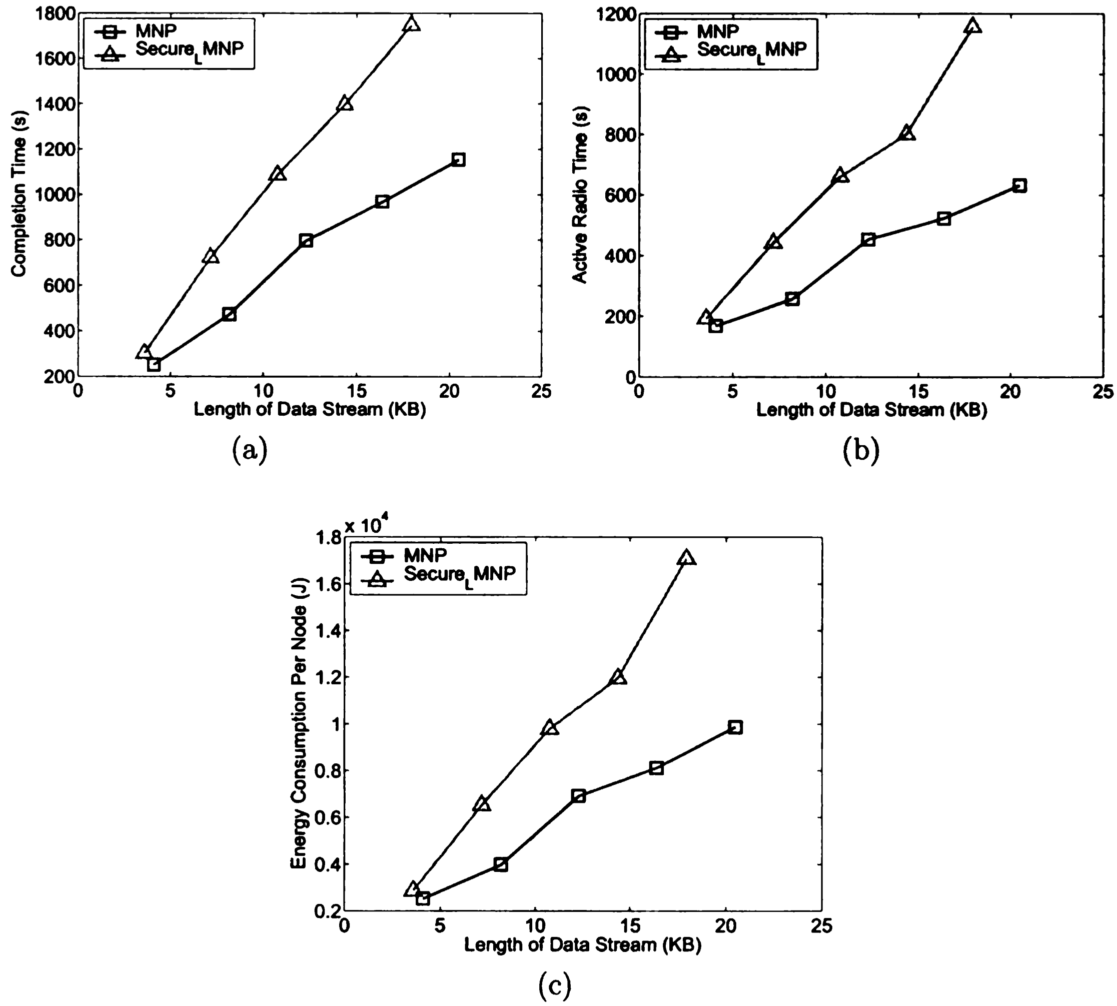
The memory requirement of our authentication protocol with the enhancements in Section 5.7 includes the amount of memory required for caching the secrets and the signatures, and that required during signing/verification process (same as what we have listed for secure reprogramming for mote-class adversaries in Section 5.5.2). In addition, the data cache and the hash cache contribute to the major part of memory consumption. If the data cache contains 8 packets, each packet is 64 bytes long (including the data part and the hashes), plus 1-byte packet ID (local ID, used inside a segment) and a *valid* bit (to indicate if the packet is in the data cache), the data cache requires 521 bytes. As we discussed in Section 5.7.2, the hash cache needs 256 bytes memory. Moreover, a variable is needed to record the current active cache group. This only needs to be 1 byte long. And, as discussed in Section 5.7.3, since the data cache provides the memory required for FEC encoding/decoding, there is no extra memory overhead by applying FEC scheme.

Performance of *Secure$_L$MNP*

We set the size of the data cache c to 8 packets. The size of FEC transmission group t is set to the same as the data cache size, which is also 8 packets. As we have pointed out earlier, 8 out of 64 bytes data payload in a data packet are used for hashes. Therefore, transmitting one segment (64 packets per segment) in MNP disseminates 4KB data, while transmitting one segment in *Secure$_L$MNP* only disseminates 3.5KB data. In Figure 5.11, we show the completion time, active radio time and energy consumption of MNP and *Secure$_L$MNP* at different data stream lengths. We can see that given a certain amount of data to transmit, the completion time required by *Secure$_L$MNP* is 37%-74% higher than that required by MNP. In both protocols, the active radio time of sensors is around 50-60% the completion time. In Figure 5.11, we show that the active radio time (and similarly, energy consumption per node) of

Secure$_L$MNP is 30%-100% higher than that of MNP. This cost is much lower than the case where we simply use the basic hash chain approach without optimization. In the latter case, the completion time of the secured version of MNP is 6 or 7 times that of the original MNP. The same conclusion holds for Deluge [23], as shown in [14,15]. Hence, the performance improvement by applying the techniques we proposed in Sections 5.7.1-5.7.3 is significant.

In Figure 5.12, we show the communication overhead of our authentication pro-



(a)

(b)

(c)

Figure 5.11: Completion time, active radio time and energy consumption of MNP and *Secure$_L$MNP*. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.

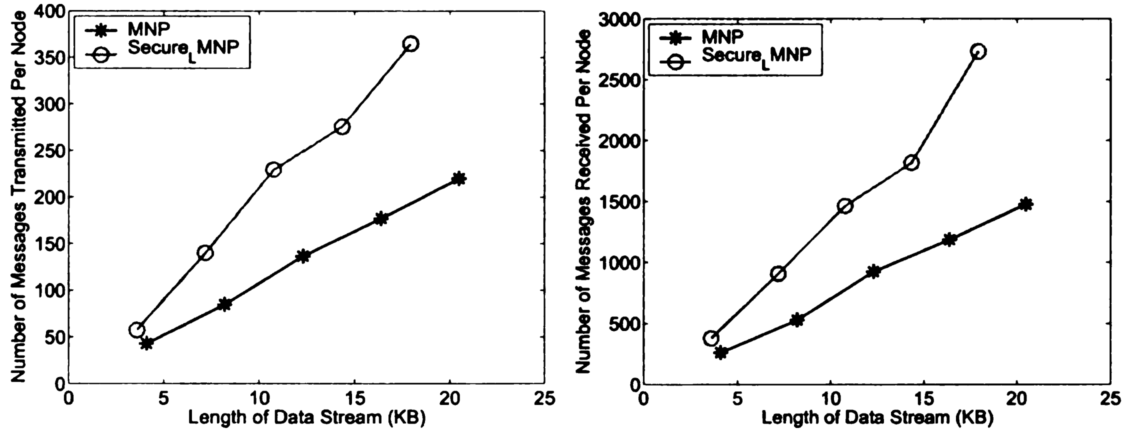
tocol. In Figure 5.12(a), we can see that for a given amount of data to be distributed, the message transmission *Secure$_L$MNP* is 52%-92% higher than that of MNP. The message reception pattern is similar to the active radio time (Figure 5.11(b)).

Analysis of Design Options

In this section, we analyze how the techniques we proposed in Sections 5.7.1-5.7.3 contribute to the overall performance. We evaluate each of them by disabling/replacing/varying it while fixing other parts of the protocol.

Comparing basic hash chain with double connected hash chain. To evaluate the effectiveness of the double connected hash chain, we compare it with the case where the basic hash chain is used (cf. Figure 5.13).

We can see that using the double connected hash chain, the performance is significantly improved compared to the basic hash chain. Replacing the basic hash chain with the double connected hash chain can reduce the completion time and the energy consumption by 73-81%. Specifically, the time (respectively, energy consumption)

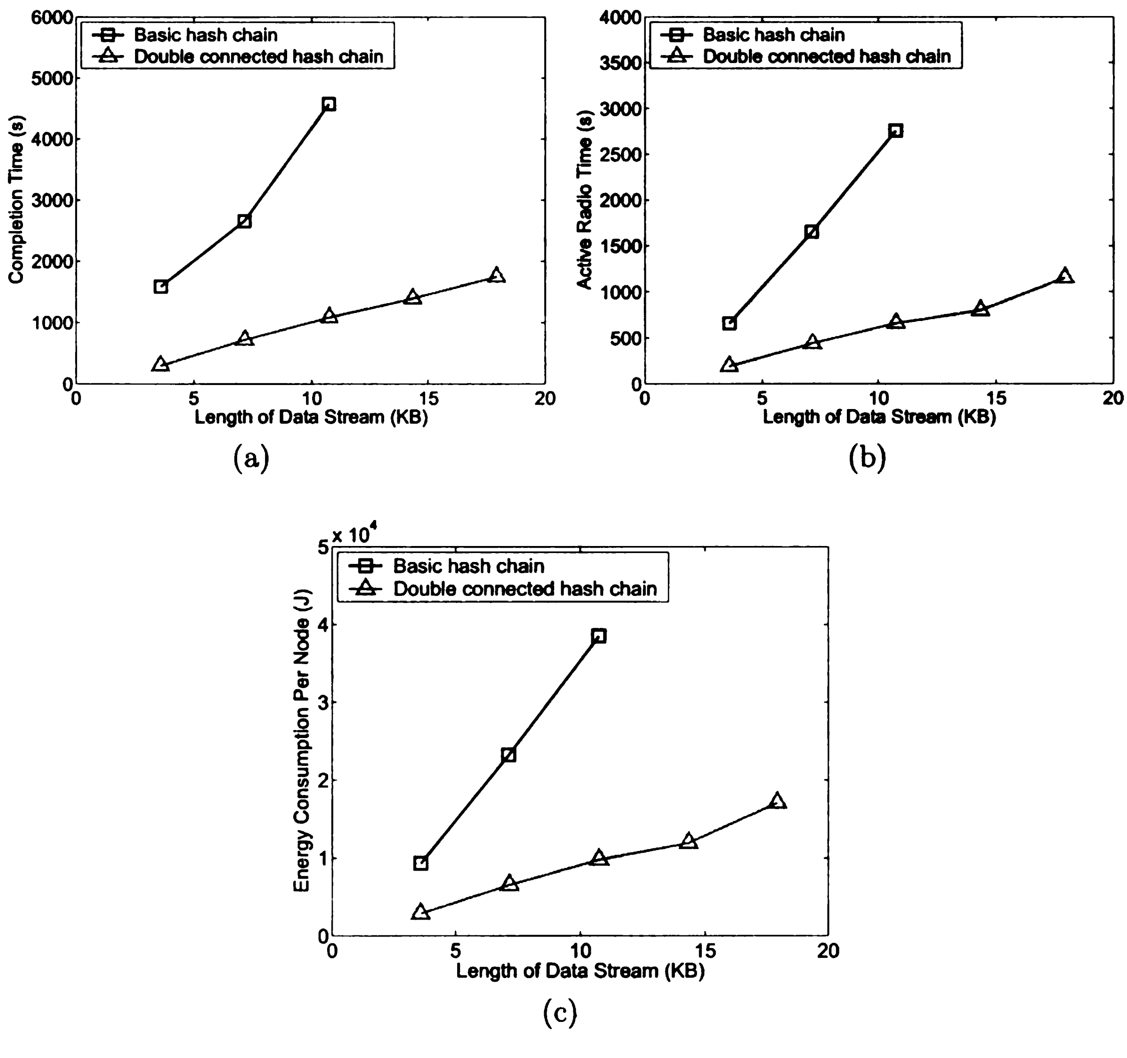


(a) (b)

Figure 5.12: Communication cost of MNP and *Secure$_L$MNP*. (a) Number of message transmitted per node vs. length of data stream (b) Number of message received per node vs. length of data stream.

to disseminate 10.7KB data using the basic hash chain scheme is 4570 seconds (respectively, 38522J), which is 4.2 times (respectively, 3.9 times) the time (respectively, energy consumption) for disseminating the same amount of data using the double connected hash chain. The message transmission and reception using the double connected hash chain are also much lower.

Varying the size of the data cache. To evaluate the effectiveness of data cache, we vary the size of the data cache from 8 packets to 64 packets. In the case



(a)

(b)

(c)

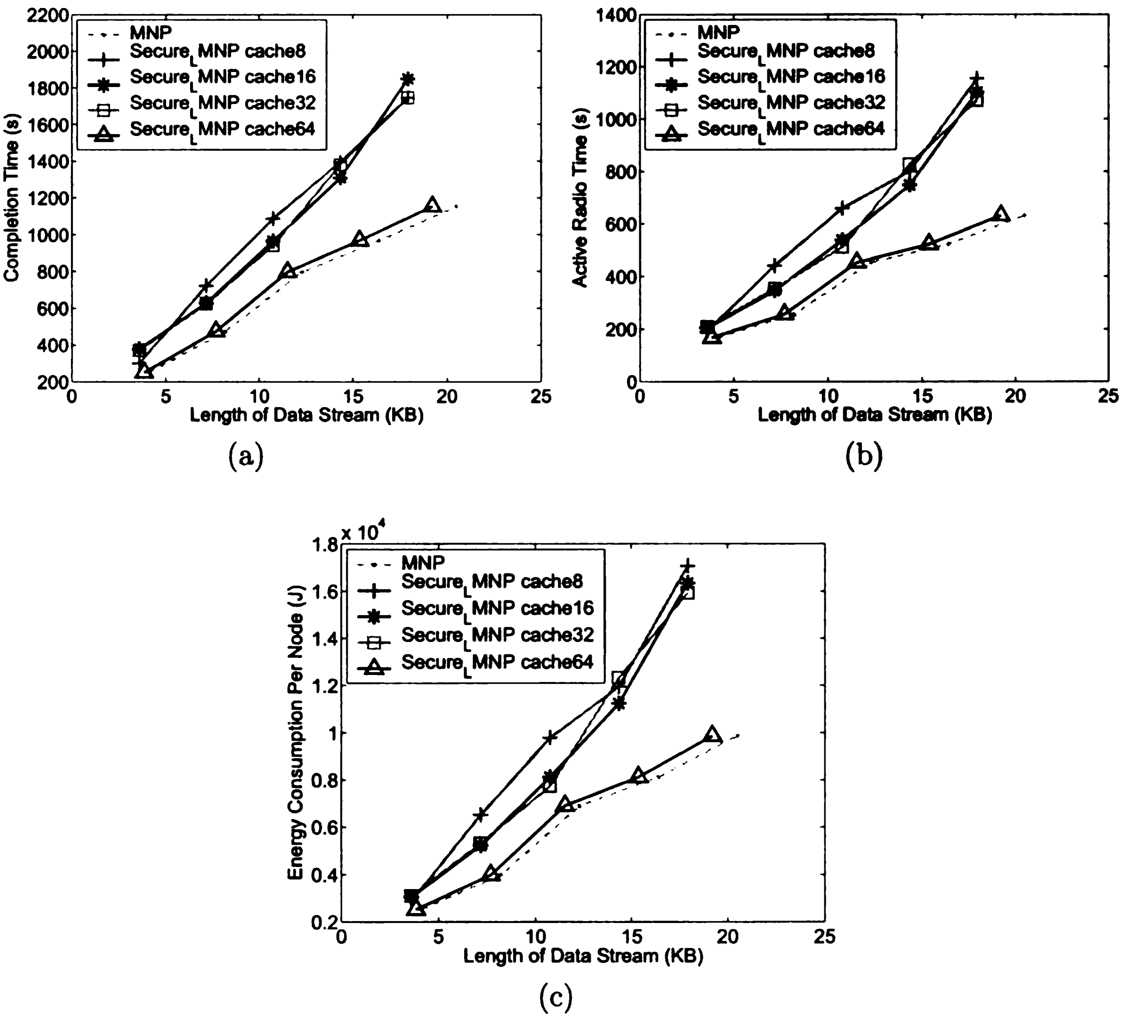
Figure 5.13: *Secure_LMNP*: comparison of the basic hash chain with the double connected hash chain. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.

that the data cache size is 64 packets, the entire segment can be stored in memory. In this case, even if the data packets arrive out of order, they can be temporarily saved in the data cache, waiting to be authenticated later. Hence, the basic hash chain works well in this scenario. In Figure 5.14, we show the completion time, the active radio time and energy consumption of *Secure$_L$MNP* for disseminating different amounts of data, when the data cache size varies. For comparison, we also show the corresponding performance of MNP. In the case that the data cache size is 64 packets, we use the basic hash chain, instead of the double connected hash chain, to reduce the cost of distributing hashes. As shown in Figure 5.14, when the data cache size is 8 packets, 16 packets, and 32 packets, the lines that represent the completion time in Figure 5.14(a) (and respectively, the active radio time and energy consumption per node in Figure 5.14(b) and (c)) intersect with each other. In other words, there is no significant performance improvement when we increase the size of the data cache.

By increasing the cache size, we are able to cache more hash groups in memory. This helps to improve the performance only if some delayed packets in the earlier group(s) arrive later. However, this long delay of packets is uncommon (although short delays of packets within a cache group are common) for two reasons: the sensors send packets in order (with increasing packet IDs), and sensors communicate with their direct neighbors (i.e., there is no path delay). Moreover, the FEC encoding/decoding is done in the active cache group. Therefore, setting the data cache size to 8 packets (which is the same as the hash group size and FEC transmission group size) is optimal (for reducing memory consumption and achieving reasonable performance), unless memory is big enough to accommodate the entire segment.

If we increase the data cache size to 64 packets, the completion time and energy consumption are reduced significantly. In this case, all the received packets are buffered in memory, the only cost is on distributing the hashes. Hence, when the data cache size is 64 packets, the completion time and energy consumption are only a little higher than that of MNP.

Effect of using FEC. Forward error correction (FEC) reduces the number of requests and retransmits by sending extra parity packets. In this section, we investigate if the use of FEC is beneficial. The size of the FEC transmission group and the size of the data cache are both 8 packets. In Figure 5.15, we compare the performance of *Secure$_L$MNP* in the cases that FEC is enabled and disabled. We can see that by employing FEC, we can reduce the completion time and energy



(a)

(b)

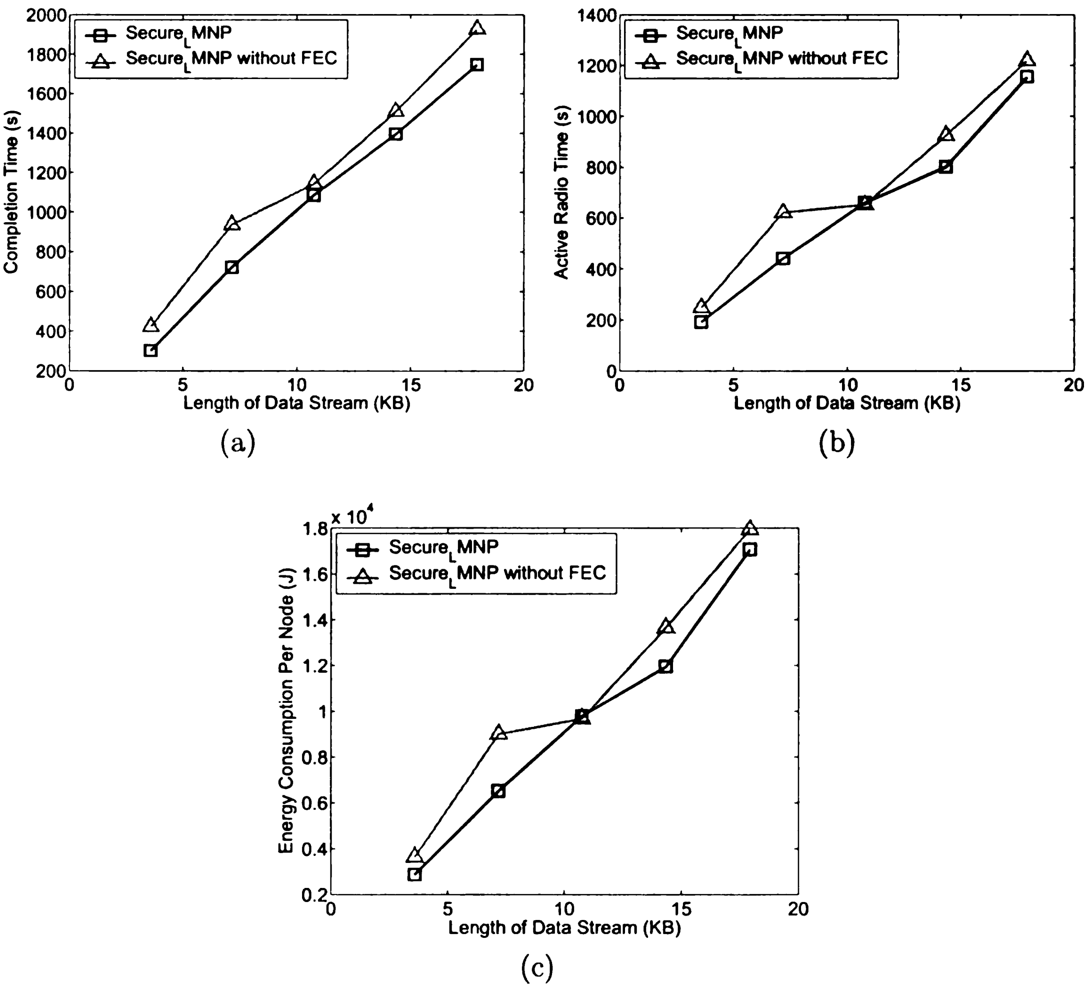
(c)

Figure 5.14: *Secure$_L$MNP*: varying the data cache size from 8 packets to 64 packets. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.

consumption by 5-29%.

In Figure 5.16, we show the message transmission and reception in the two cases, *Secure$_L$MNP* with FEC enabled (the default case) and with FEC disabled. We can see that using FEC also reduces message transmission and reception.

We change the size of the data cache to 16 packets and 32 packets, and repeat the same simulation. From Figures 5.17 and 5.18, we can see that using FEC can reduce the completion time and energy consumption in both cases.

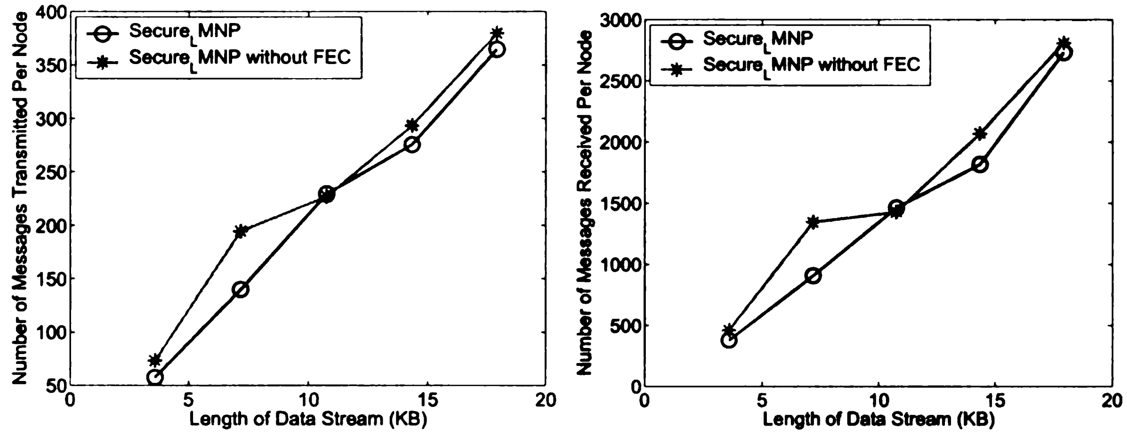


(a)

(b)

(c)

Figure 5.15: *Secure$_L$MNP*: effect of using FEC. The data cache size is 8 packets. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.



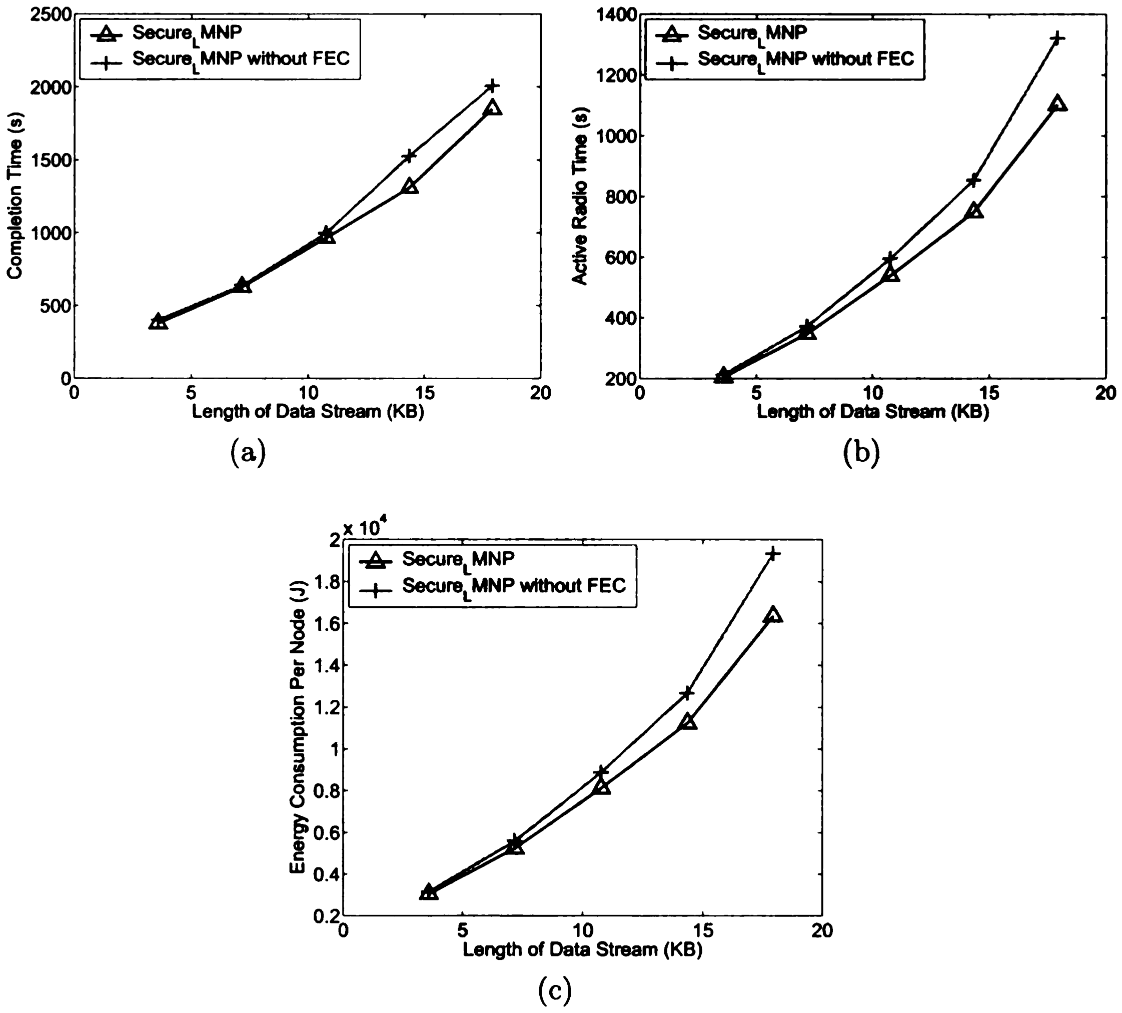
(a) (b)

Figure 5.16: *Secure$_L$MNP*: effect of using FEC (communication overhead). The data cache size is 8 packets. (a) number of messages transmitted per node vs. length of data stream (b) number of messages received per node vs. length of data stream.

5.8 Discussion: Key Distribution and Updates

In this section, we discuss the issues of initial key distribution and key updates. As discussed in Section 5.4, the initial keys are assigned to sensors at deployment. Alternate approaches are also possible for distribution. For example, at deployment, the sensors may include all the secrets and then depending upon their logical ID (either known at deployment or communicated thereafter), they can delete the secrets they are not supposed to have. This approach is based on the assumption (true in many sensor network deployments) that the initial communication among sensors is secure and that the sensors cannot be compromised for a certain duration after deployment.

As discussed in Section 5.4, the approach in [18,31] allows us to provide a tradeoff between the level of security and the number of secrets maintained by the base station. The designer can choose an appropriate value of r to ensure that the effect of collusion is moderate. Increasing the value of r increases the overhead only marginally, as the cost of signing with a symmetric key is very small.

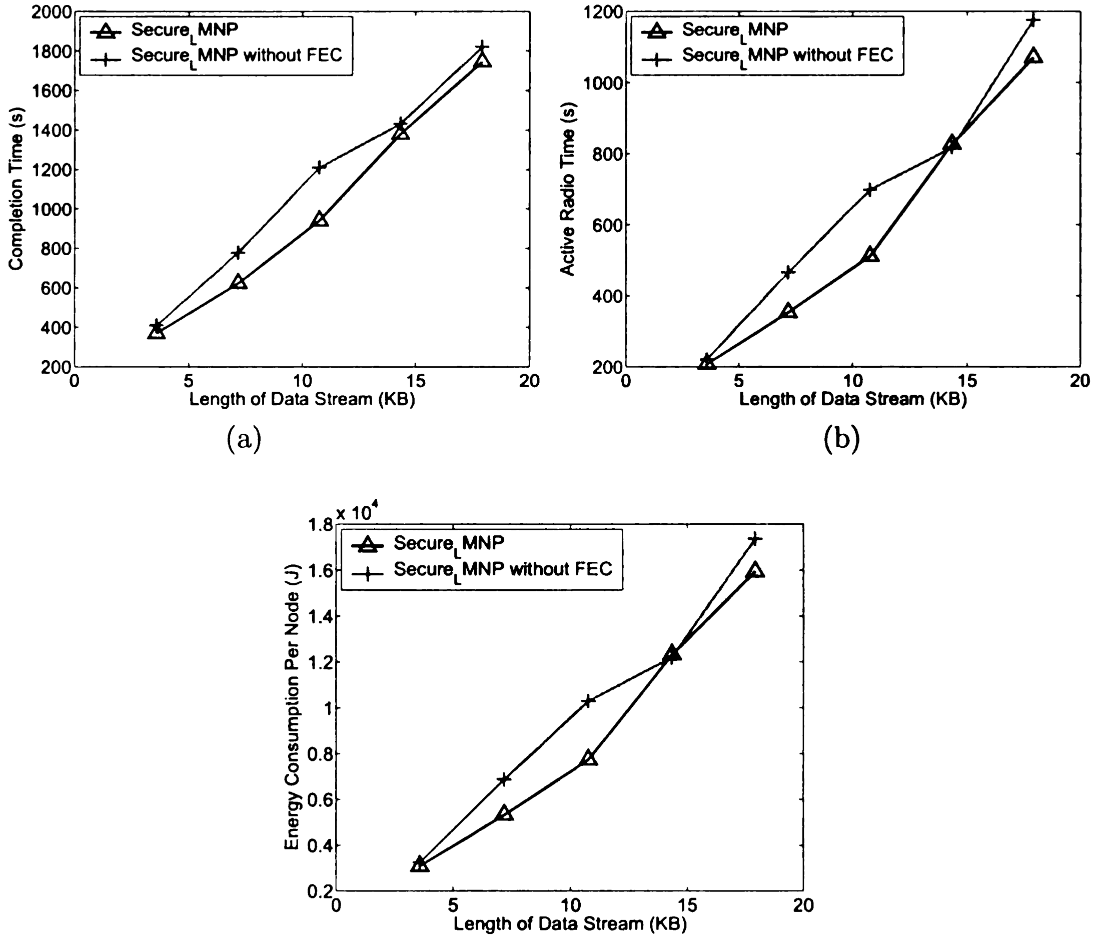


(a)

(b)

(c)

Figure 5.17: *Secure$_L$MNP*: effect of using FEC. The data cache size is 16 packets. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.



(a)

(b)

(c)

Figure 5.18: *Secure$_L$MNP*: effect of using FEC. The data cache size is 32 packets. (a) completion time vs. length of data stream (b) active radio time vs. length of data stream (c) energy consumption per node (not including the authentication cost) vs. length of data stream.

Also, our approach can be combined with the approach in [40,41,51]. In particular, instead of maintaining a single value for each secret, we could have a hash chain of values for each secret. (Optimizations from [6] allow only a little more than $\log_2 n$ entries from a hash chain to be maintained at the base station.) The last value in the hash chain is made available to the sensors at the time of deployment. However, periodically, the base station will reveal the previous value in the hash chain and this value would be encrypted with the current secret to ensure that only those sensors that have the old secret can obtain the new one.

5.9 Chapter Summary

In this chapter, we showed how authentication could be achieved for reprogramming, and more generally, bulk data dissemination, in sensor networks. We used symmetric key distribution algorithms from [18,31] to ensure that the base station can communicate securely with each sensor in the network. Based on the security of the key distribution, our protocol allows sensors to conclude that the data is truly transmitted by the base station.

We first focused on the case where only mote-class adversaries exist. Since such adversary has limited energy, it cannot use extensive denial of service attack. our algorithm is expected to be especially valuable for security in sensor network testbed. Such testbed is typically physically secure, thereby preventing/mitigating laptop-class attackers. However, the testbed typically relinquishes control of individual sensor nodes that are used in an experiment. Thus, an experiment could be interfered by other sensors in the testbed. Our algorithm provides protection from such interference/attacks with a low overhead.

We then provided solution for the case where laptop-class adversaries exist. A laptop-class adversary can mount a denial of service attack by sending garbage data to the motes. To mitigate denial of service attacks, we require that the sensors will not save any packets to EEPROM (an energy consuming operation) unless the packet is authenticated. We considered the secure data dissemination problem in three scenarios: multihop dissemination with coarse-grained (segment-level) pipelining, multihop dissemination with fine-grained (packet-level) pipelining, and single-hop dissemination. We showed that the use of symmetric keys can significantly reduce the cost of secure dissemination of a moderate amount of data, especially in the second and third scenarios. In particular, the time to sign and verify a single packet using public key scheme is more than 22 seconds. Within the same amount of time, we can disseminate 3-4KB of data across the network using symmetric key scheme. For the first scenario, we proposed additional mechanisms to reduce the cost of secure data dissemination. We showed that the basic hash chain that is commonly used for authenticating data streams is not efficient in the presence of packet loss, as it requires that packets arrive in order. We proposed the double connected hash chain to strengthen the inter-connection among the packets so that loss of a few packets does not fail the authentication for the entire segment. To further improve the performance, we proposed a caching scheme and employed forward error correction (FEC) technique to try to minimize the effect of packet loss.

We showed that our algorithm can be easily applied to different types of data dissemination protocols: simple non-pipelined, fine-grained pipelined (e.g., Infuse [30], Sprinkler [47]), and coarse-grained pipelined (e.g., MNP (cf. Chapter 2), Deluge [23]). Hence, our protocol is applicable to various application scenarios. We analyzed/simulated the overhead of our protocol, and showed the effectiveness of our design in enhancing the performance.

As discussed in Section 5.4, the key distribution algorithm allows the designer to choose the appropriate parameter, r, to determine the desired level of collusion resistance. With the use of this parameter, the base station maintains $rlog_r n$ (n

is the number of sensors) secrets and each sensor maintains $log_r n$ secrets. With increased value of r, the collusion resistance increases, and each sensor maintains fewer secrets. As a tradeoff, the base station maintains more secrets, and the cost of creating/verifying the signatures increases. For example, in a 10x10 network, if we increase r from 2 to 10, the number of secrets maintained at the base station increases from 14 to 20. However, due to the facts that the symmetric key operation is very fast (e.g., as discussed in Section 5.3, the execution time of encrypting/decrypting a 8-byte block using RC5 is only 0.26ms) and these secrets are used only a few times during data dissemination, this increase in the cost of signing/verification is negligible. Hence, the performance of data dissemination changes minimally when we increase the level of collusion resistance.

Chapter 6

Related Work

In this chapter, we discuss work in the areas of network reprogramming (Section 6.1), FEC coding schemes (Section 6.2), and secure reprogramming (Section 6.3).

6.1 Network Reprogramming

Reprogramming Protocols. The existing work on delivering the entire program to all the sensors in the network includes TinyOS single-hop network reprogramming (XNP) [12] and multihop network reprogramming approaches, such as MOAP (Multihop Over-the-Air Programming) [60], Deluge [23], MNP (cf. Chapter 2), Infuse [30], and Sprinkler [47]. All these approaches assume that one (or a few) sensor has the entire new program initially, and communicates the new program to the remaining sensors in the network. By contrast, *Gappa* (cf. Chapter 3) is designed for the scenario where some sensor nodes have received one segment initially, and communicate with each other to receive the remaining segments.

In contrast to MNP and *Gappa*, XNP, MOAP, and Deluge do not turn off sensors' radio during reprogramming. We have compared the performance of Deluge with that of MNP and *Gappa* in Sections 2.2.2 and 3.3.

MOAP is a multihop network reprogramming approach. MOAP disseminates code in a hop-by-hop fashion, that is, a node has to receive the entire program image before starting advertising. MOAP uses a simple publish-subscribe interface for reducing the number of senders. No sender selection mechanism is considered. If a loss is detected, a NAK is unicast to the sender requesting for retransmission. To keep track of loss information, a sliding window approach is proposed.

MNP and ***Gappa***, and many other reprogramming protocols we have mentioned (XNP, MOAP, and Deluge), use CSMA-based MAC protocol. Infuse [30] and Sprinkler [47] are two TDMA-based reprogramming protocols. A TDMA-based protocol provides the advantages that a node transmits messages only in its assigned time slots, so that message collision is avoided and the node can turn off its radio when it is not transmitting or receiving. However, TDMA requires the time synchronization service, and it is designed for algorithms where topology is known upfront.

To address the very resource constrained nature of sensor nodes, Maté [35] is included in TinyOS. Maté is a stack-based virtual machine. Programs are represented as one or a few *capsules* (current implementation allows at most eight capsules), of up to 24 instructions. Each capsule fits in a packet and can be propagated to other nodes. In this way, Maté allows new programs to be forwarded and installed quickly through a network. This virtual machine approach is complementary to the entire code image delivery approaches. For example, we might need to delivery the binary code of the virtual machine itself to sensor nodes.

The Firecracker protocol [36] proposed by Levis and Culler is designed to deliver *small pieces* of data from one or a few sources to every node in a network. It uses a combination of routing and broadcasts to speed up dissemination. Data is first routed to several distinct points (seed points) in the network; once data arrives, broadcast-based dissemination starts from the destinations. The authors conclude that increasing the number of seed points and selecting the seed points that are distant improve performance. Our simulations also show that using more than one base stations and placing them far away from each other (at corners) can speed up reprogramming by 13-24% compared to the case where one base station is placed at a corner.

Suppression schemes. *Message implosion* or *broadcast storm* problem [48] exists in both wired and wireless networks. Suppression schemes normally fall into two categories: aggregation based, deferred feedback based. Aggregation based suppression is usually used in large sensor networks. Data is aggregated at intermediate nodes on the way to the destination node. This approach, called ***in-network** aggregation*, was proposed in Directed Diffusion [24], and broadly used in almost all flat structured or cluster-based protocols, such as LEACH [21], SINA [57].

In deferred feedback based suppression, each node defers its sending of response for a certain period of time, during which it may cancel its response if it hears an identical one from its neighbors, or it may send response probabilistically based on the number of identical replies it has heard. Two examples of deferred feedback based suppression are *Scalable Reliable Multicast* (SRM) [16] and Trickle [39]. Trickle dynamically adjusts the advertise interval so that it propagates code rapidly during the active updating phase, and has low overhead during maintenance phase. We use a similar idea in MNP to dynamically scales the noreq wait period.

The sender selection algorithm in MNP and ***Gappa*** is also delay based. We use "number of requesters" as the criteria to choose sender. The goal is to find the "good" senders who have many "followers".

Multi-Channel Reprogramming The only work on multi-channel reprogramming we are aware of is [70], where the authors present preliminary experiment results (based on 25 nodes). In [70], the authors propose an algorithm, *Multi-Channel Deluge*, which divides nodes into *groups* based on node ID or geographically, and assigns a channel to each group. Similar to other existing reprogramming approaches, it also assumes that one or a few source nodes have the complete new program, and disseminate the new program to the entire network. In the algorithm proposed in [70], there are specially marked nodes in group 1 (the default group), which form a connected dominating set, so that all the nodes in the network can directly communicate to at least one node belonging to group 1. By contrast, in ***Gappa***, all the nodes are equal. Hence, there is no dependency on special nodes.

6.2 Forward Error Correction (FEC)

Automatic repeat request (ARQ) and forward error correction (FEC) are the two basic ways to provide reliability for transmission protocols. All the existing work on network reprogramming [12,23,30,33,47,60,66] uses ARQ-based approaches for error recovery. We propose adding FEC to the ARQ-based reliability scheme, and perform a case study on MNP.

There are different types of FEC codes. We have introduced simple XOR code and Reed-Solomon (RS) codes in Section 4.2.1. Both XOR code and RS codes belong to block (n, k) FEC codes. A block code has the property that any k out of the n encoding packets can reconstruct the original k source packets.

Tornado codes [43] provide an alternative to RS codes. Tornado codes have lower computation complexity than RS codes, at the small cost of reception overhead, that is, a (n, k) Tornado code requires slightly more than k out of n encoding packets to recover k source packets.

Unlike the block (n, k) codes, Luby Transform (LT) codes [42] can generate as many unique encoding packets as required, using the k source packets as input. Each encoding packet is generated randomly and independently of all other encoding packets. LT codes have the property that the receiver is able to reassemble the original k source packets as long as it receives enough number (slightly more than k) of encoding packets. LT codes are designed for delivering a large amount of data over high bandwidth internet links. They have lower computation complexity on encoding and decoding than RS codes. However, they introduce higher recovery overhead because more redundant packets are transmitted. Normally, the number of repair packets are more than 10 times the number of source packets.

6.3 Secure Data Dissemination

Security of data dissemination in sensor networks is studied in [5, 14, 15, 26, 27, 34, 40, 41, 49, 51, 56]. The two protocols, Sluice [34], proposed by P. Lanigan *et. al.*, and SecureDeluge [15], proposed by P. Dutta *et. al.*, use the basic hash chain for authentication. Sluice verifies hashes at the segment level, while SecureDeluge verifies hashes at the packet level. Although segment-level hash chain has low computation, communication and memory cost, Sluice is vulnerable to some form of attacks, e. g., a single corrupted/lost packet will cause the entire segment to be discarded. The problem with the basic hash chain approach, as we discussed, is that it requires that packets be received/stored in order. All the packets that arrive out of order are thrown away. This requirement increases the delay and message cost significantly, especially when the network is lossy. There is another protocol proposed by J. Deng *et. al.* [14], which tries to address the problem by sending a hash tree over the data packets before sending the actual data packets. After sensors have received the entire hash tree, they can receive/verify data packets that arrive out of order.

All these three protocols mentioned above [14, 15, 34] use asymmetric keys; the base station signs the (hash of) data using the private key and the sensors use the corresponding public key to authenticate the data. As discussed in Section 5.6, the cost of asymmetric keys is exorbitant when the data size is moderate (1-4KB). In particular, in Section 5.6, we showed that for several scenarios of moderate data dissemination, the cost of signing and sending one packet using asymmetric keys is close to the cost of sending 3-4 KB of data using symmetric keys. Finally, our approach is orthogonal to that in [14], i.e., as discussed in Section 5.6.3, our approach of using symmetric keys could be used in [14] to reduce the cost of those algorithms.

A. Perrig *et. al.* proposed TESLA [50] and μTESLA [51] to provide broadcast authentication through a hash chain. μTESLA is designed to work on the resource-constrained sensor nodes. It applies symmetric keys, and achieves asymmetry for authentication by delaying the disclosure of the symmetric keys. Liu and Ning proposed a multi-level μTESLA [40] to extend the capability of μTESLA. By constructing multi-level μTESLA structure and using higher-level μTESLA instances to authenticate the parameters of lower-level ones, this extension enables the original μTESLA to cover a long time period and support a large number of receivers. In their follow-up work [41], Liu *et. al.* improved the μTESLA to support a large number of broadcast senders and mitigate denial of service attacks by distributing the μTESLA parameters using a Merkle hash tree [46].

Our approach differs from μTESLA [51] and its extensions [40,41] in that in [40, 41,51], *after-the-fact* authentication is provided. In particular, in these approaches, the sensors first receive a packet (set of packets) and subsequently receive the key to authenticate it. To ensure security of such a protocol, the time to reveal the key must be large enough so that all nodes receive the packet (respectively, set of packets) before the key is revealed (loose time synchronization is required in these protocols). In the context of bulk data transmission, this would require the sensors to buffer a large amount of data before it can be authenticated. With limited storage on sensors, this would require the data to be stored on EEPROM, thereby, opening up the possibility of denial of service attack. By contrast, in our approach, only authenticated packets are stored on EEPROM. Thus, the approaches in [40,41,51] are applicable in broadcasts of small data whereas our approach is also applicable in broadcasts of bulk data as well. Moreover, as discussed later in this section, it is possible to combine the features of our algorithm and that in [51].

One-time signatures commit a secret key via one-way functions, hence, have much lower signing and verification time compared to asymmetric primitives. One-time signatures have been used in many broadcast authentication protocols [5,26,27,49,56]. BiBa [49] performs authenticated broadcast via pre-computed hash collisions and chains. HORS [56] improves BiBa by reducing the signature generation time. BiBa and HORS are inappropriate for sensor networks due to its large public key size (e.g., a typical public key size is 20KB). Since the public key must be stored on all sensors, it is desirable to keep its size small. The approach in [5,26,27] decreases the public key size of HORS by constructing Merkle trees [46] on the secret keys, and use the roots of the Merkle trees as the public key. By varying the number of trees, this approach is able to tradeoff public key size for signature size. Even with this reduction, the public key size is still hundreds to thousands of bytes, which is much larger compared to the size of secrets (e.g., at most 7 secrets in a 10x10 network) in our protocol. Moreover, the signature size in [5,26,27] is also large. For example, in [27], the typical signature size is 690-2560 bytes. Hence, when the authors apply their one-time signature protocol to Deluge [23], the first few *pages* are used for sending the signature. By contrast, in our protocol, all the signatures are sent in one (or a few) message(s). Therefore, our protocol has much lower memory requirement and communication overhead.

Chapter 7

Conclusion and Future Research

Reprogramming sensor networks in place via radio is an essential service due to the facts that sensor networks consist of hundreds or thousands of sensor nodes and they are often deployed in remote or hostile environments. Reprogramming for sensor networks requires 100% reliability. And, due to the extreme resource constraints of sensor nodes, it is necessary to reprogram sensors in a fast and energy efficient way. Moreover, reprogramming is vulnerable to packet injection and corruption, it is important that sensors be able to verify that the code image is from a trusted source. Hence, providing authentication for sensor networks is necessary.

In this dissertation, we proposed two reprogramming protocols, which are designed for different scenarios. In the first case, one (or a few) sensor has the entire new program. It, then, propagates the program to all the sensors in the network. In this model, all the sensors communicate on a single shared radio channel. In the second case, each sensor (or a subset of sensors) has one part of the program initially. The sensors then communicate with each other to receive the remaining parts of the program using multiple radio channels. We proposed MNP for the first reprogramming model, and *Gappa* for the second model.

To reduce message collision, MNP uses a sender selection algorithm to try to guarantee that only one sensor is transmitting in a neighborhood at a time. In MNP, sensors receive the program segments in order. The sensors advertise the highest segment ID it has, as well as the number of (distinct) requests it has received. The sender selection algorithm selects the sensors that have received the most number of requests, and it gives priority to the segments with lower ID. If a sensor loses in the sender selection algorithm and it is not interested in receiving the code from its neighbors, it goes to sleep. In this way, MNP effectively reduces the active radio time of sensors and saves energy. MNP allows the sensors at different neighborhoods (their transmissions do not collide) to transmit simultaneously. This reduces reprogramming time.

Gappa extends the sender selection algorithm to multiple channels so that on each channel, at most one sensor in a neighborhood transmits the code at a time. *Gappa* allows sensors to receive program segments out of order. The sensors communicate the segments they have on the control channel. If a sensor is selected as a sender, it transmits the data on a data channel. If a sensor loses in the sender selection algorithm on one channel, it will try to transmit on a different channel. If all the channels are busy and the sensor does not intend to receive code from any of its neighbors, it goes to sleep. Similar to MNP, *Gappa* also saves sensors' energy by trying to eliminate idle listening, which is the major energy waste on sensors.

We proposed two ways to select a data channel in *Gappa*. Fixed channel allocation assigns a fixed data channel to each segment, while variable channel allocation allows a sensor to select a data channel randomly among all the available channels. We show that variable channel allocation performs better than the simple fixed channel allocation as it maximizes channel utilization and enables higher concurrency.

We showed that due to the use of multiple radio channels and gossip based communication (allowing sensors to receive segments out of order), *Gappa* is able to reprogram a sensor network faster and with lower energy cost, compared to the protocols proposed for the first reprogramming model, such as MNP and Deluge [23]. However, *Gappa* and MNP are not replacements of each other as they are designed for different reprogramming scenarios. It is important to choose the appropriate reprogramming protocol according to the real world condition (e.g., initial distribution of data, availability of multiple channels, etc.).

MNP and *Gappa*, as well as existing reprogramming protocols (e.g., [23, 30, 47, 60]), provide reliability through ARQ based mechanism. We showed that using FEC codes can effectively reduce packet loss, and hence, reduce reprogramming time and energy consumption. We applied two block FEC codes, simple XOR code and Reed-Solomon (RS) codes, to MNP, and studied the tradeoff between the computation cost and performance improvement. Depending on the amount of computation resources (e.g., processor, memory) available on sensors, we can either use a simple scheme (such as XOR code) for limited improvement, or choose a more powerful coding scheme (such as RS codes) for better performance.

We also proposed a symmetric key based protocol for authenticating reprogramming process. Our protocol is based on the secret instantiation algorithm from [18,31], which ensures that the base station can communicate securely with each sensor in the network. We first focused on the case where only mote-class adversaries exist (i.e., there is no denial of service attack). We showed that our protocol is able to provide authentication at very low communication cost, and has very short delay and small memory footprint. We then provided solution for the case where laptop-class adversaries exist. To mitigate the denial of service attacks from a laptop-class adversary, we require that the sensors will not save any packets to EEPROM (an energy consuming operation) unless the packet is authenticated. We showed that by using symmetric keys, our protocol significantly reduces the cost of securely disseminating a moderate amount of data, compared to public key based authentication protocols. In particular, the time to disseminate 3-4 KB data using symmetric keys is the same as the time to sign and verify a *single* packet using a public key scheme. We also propose additional techniques to further reduce the authentication cost when the program/data size is large. We illustrated our protocol in the context of various types of reprogramming protocols, including MNP (CSMA-based protocol using coarse-grained pipelining) and Infuse [30] (TDMA-based protocol using fine-grained pipelining). Moreover, we showed that our authentication protocol is applicable not only to reprogramming, but also to other bulk data dissemination scenarios such as network monitoring, difference-based reprogramming.

The research presented in this dissertation can be extended in the following directions:

- *Comprehensive power management that maintains sensing coverage and connectivity during reprogramming.* In our recent work [67] (not included in this dissertation), we proposed a simple, local protocol, *pCover*, that maintains partial (but high) coverage while turning off the redundant sensors to save energy. Through *pCover*, we demonstrated the tradeoff between sensing coverage and network lifetime. We showed that it is feasible to maintain a high partial coverage ($\sim$ 90%) while significantly increasing network lifetime when compared with protocols that provide full coverage. We note that the power management protocols such as *pCover* are used when a network is working (i.e., not in maintenance/reprogramming state). One of the assumptions in all the existing reprogramming models is that when reprogramming starts, the network stops functioning. However, in some situations, it is desirable that the service interrupt duration is minimized. In this case, the network should continue working (e.g., monitoring the environment) even *during* reprogramming. MNP and *Gappa* conduct power management through the sender selection algorithm, which turns off all the sensors that are not actively transmitting or receiving the new program. Such issues like sensing coverage, network connectivity, are not considered. In order to allow the network to continue functioning during reprogramming, we need to provide a new power management technique that satisfies the requirements of reprogramming and at the same time maintains the desired degree of sensing coverage and network connectivity.

- *Gossip based data collection.* *Gappa* is designed for gossip based data dissemination. A related problem is gossip based data collection. The major differences between gossip based data dissemination and data collection are as follows. First, in data collection, each sensor (or a subset of sensors) is associated with some data. The total number of data elements in data collection tends to be larger than the number of data segments in data dissemination. Second, during data collection, in most cases, sensors do not simply store and forward the data they have received. Rather, they could perform aggregation on the collected data and advertise a smaller number of (aggregated) segments. Consider an example that one sensor is advertising the aggregation of segments 1-5, and the other sensor is advertising the aggregation of segments 3-7. In other words, the segments that the sensors are advertising are overlapping. The receivers need to decide which parts contain new information and how to perform additional aggregation based on the received aggregated segments.

- *Hybrid protocol that combines MNP and Infuse [30].* We note that one reason that *Gappa* performs better than MNP is that the code segments are distributed in different parts of the network in the initial state, which allows sensors in different locations to start communicating code immediately when reprogramming starts. We have already shown that using more than one base stations and placing them far away from each other (at corners) can reduce the reprogramming time by 13-24% compared to the case where one base station is placed at a corner (cf. Section 2.2.2). The Firecracker protocol [36] also suggests that sensors first route data to several distinct points (seed points) in the network, then start broadcast-based dissemination, can improve the performance of data dissemination. However, the Firecracker protocol is designed for disseminating *small pieces* of data. In the context of reprogramming, if we use a coarse-grained pipelining protocol such as MNP, *Gappa*, or Deluge [23], the cost of disseminating a segment to a point that is many hops away from the base station is high. We can instead use a TDMA based protocol such as Infuse [30] to distribute different code segments to the sensors that are far away from the base station in a fine-grained pipelining fashion. The fine-grained pipelining protocols (e.g., Infuse) have lower cost in disseminating a moderate amount of data over a long distance compared to the coarse-grained pipelining protocols (e.g., MNP, *Gappa*, Deluge). After these initial segments have arrived to those remote sensors (or seed points), we can start reprogramming using the protocols such as MNP (or *Gappa*, Deluge). We expect that this will improve reprogramming performance if the network is large. The selection of the seed points will also affect the performance.

Bibliography

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5):605–634, December 2004.

[2] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, M. Gouda, Y. Choi, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker. Exscal: Elements of an extreme scale wireless sensor network. *The International Conference on Real-Time and Embedded Computing Systesm and Applications (RTCSA)*, August 2005.

[3] M. Arumugam, L. Wang, and S. S. Kulkarni. A case study on prototyping power management protocols for sensor networks. *In The 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, November 2006.

[4] S. Bapat, V. Kulathumani, and A. Arora. Analyzing the yield of exscal, a large-scale wireless sensor network experiment. *the 13th IEEE Symposium on Reliable Distributed Systems (SRDS)*, November 2005.

[5] S. M. Chang, S. Shieh, W. W. Lin, and C. M. Hsieh. An efficient broadcast authentication scheme in wireless sensor networks. *The 2006 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, March 2006.

[6] D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. *The Fifth Conference on Financial Cryptography (FC)*, February 2002.

[7] Crossbow Technology, Inc. *MICA2 Datasheet.* Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datas%heet.pdf`.

[8] Crossbow Technology, Inc. *MICAz Datasheet.* Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datas%heet.pdf`.

[9] Crossbow Technology, Inc. *MPR-MIB Users Manual, Revision B, June 2006, PN: 7430-0021-07.* Available at:`http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_M%anual.pdf`.

[10] Crossbow Technology, Inc. *MSP410 Datasheet.* Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MSP410_Data%sheet.pdf`.

[11] Crossbow Technology, Inc. *TELOS-B Datasheet.* Available at: `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Data%sheet.pdf`.

[12] Crossbow Technology, Inc. *Mote In-Network Programming User Reference Version 20030315*, 2003. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[13] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured flash file system for micro sensor nodes. *The 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys)*, November 2004.

[14] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks. *the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, April 2006.

[15] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the deluge network programming system. *the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, April 2006.

[16] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 12 1997.

[17] R. Gennaro and P. Rohatgi. How to sign digital streams. *Lecture Notes in Computer Science*, 1294:180+, 1997.

[18] M. Gouda, S. S. Kulkarni, and E. Elmallah. Logarithmic keying of communication networks. *In Proceedings of The Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems*, November 2006.

[19] C. Gui and P. Mohapatra. Power conservation and quality of surveillance in target tracking sensor networks. *In Proceedings of the Tenth Annual International Conference on Mobile Computing and Networking (ACM MobiCom)*, October 2004.

[20] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. *the 6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES04)*, August 2004.

[21] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. *In Proceedings of the 33rd Hawaii International Conference on System Sciences*, Janauary 2000.

[22] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *The Ninth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-IX)*, pages 93–104, November 2000.

[23] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the second International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, Baltimore, Maryland, 2004.

[24] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[25] C. Karlof, N. Sastry, and D. Wagner. Tinysec: A link layer security architecture for wireless sensor networks. *the 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys)*, November 2004.

[26] I. Krontiris and T. Dimitriou. Authenticated in-network programming for wireless sensor networks. *The 5th International Conference on AD-HOC Networks and Wireless (Adhoc-Now)*, 2006.

[27] I. Krontiris and T. Dimitriou. A practical authentication scheme for in-network programming in wireless sensor networks. *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2006.

[28] J. Kulik, W. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8:169–185, 2002.

[29] S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In *Sensor Network Operations*. IEEE Press, 2005.

[30] S. S. Kulkarni and M. Arumugam. Infuse: A tdma based data dissemination protocol for sensor networks. *International Journal on Distributed Sensor Networks (IJDSN)*, 2(1):55–78, 2006.

[31] S. S. Kulkarni and M. G. Gouda. A note on instantiating security in sensor networks. Available at `http://www.cse.msu.edu/~sandeep/securitydistribution/`.

[32] S. S. Kulkarni, M. G. Gouda, and A. Arora. Secret instantiation in ad hoc networks. *Special Issue of Elsevier Journal of Computer Communications on Dependable Wireless Sensor Networks*, 2005.

[33] S. S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. *In Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, pages 7–16, June 2005.

[34] P. E. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. *the 26th International conference on distributed computing systems (ICDCS 06)*, July 2006.

[35] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, 2002.

[36] P. Levis and D. Culler. The firecracker protocol. *In Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.

[37] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, Los Angeles, CA, November 2003.

[38] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[39] P. Levis, N. Patel, S. Shenker, and D. Culler. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. Technical report, University of California at Berkeley, 2003.

[40] D. Liu and P. Ning. Multi-level μtesla: Broadcast authentication for distributed sensor networks. *ACM Transaction in Embedded Computing Systems (TECS)*, 3(4):800–836, November 2004.

[41] D. Liu, P. Ning, S. Zhu, and S. Jajodia. Practical broadcast authentication in sensor networks. *The 2nd Annual International Conference on MObile and Ubiquitous Systems: Networking and Services (MobiQuitous)*, pages 118–129, July 2005.

[42] M. Luby. LT codes. *In Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[43] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory, Special Issue: Codes on Graphs and Iterative Algorithms*, 47(2):569–584, February 2001.

[44] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[45] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. *the 1st IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[46] R. C. Merkle. Protocols for public key cryptosystems. *IEEE Symposium on Research in Security and Privacy*, pages 122–134, April 1980.

[47] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. *To appear in Proceedings of the 26th IEEE Real-Time Systems Symposium*, December 2005.

[48] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Seattle, Washington, August 1999.

[49] A. Perrig. The biba one-time signature and broadcast authentication protocol. *Proceedings of the Eighth ACM Conference on Computer and Communication Security (CCS-8)*, November 2001.

[50] A. Perrig, R. Canetti, J. Tygar, and D. X. Song. Efficient authentication and signing of multicast streams over lossy channels. *IEEE Symposium on Security and Privacy*, pages 56–73, May 2000.

[51] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. *Seventh Annual International Conference on Mobile Computing and Networks (MobiCOM 2001)*, July 2001.

[52] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[53] J. Polastre, R. Szewcyzk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. *In Proceedings of the 16th Symposium on High Performance Chips (HotChips)*, August 2004.

[54] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[55] I. S. Reed and G. Solomon. Ploynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(10):300–304, 1960.

[56] L. Reyzin and N. Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. *The 7th Australian Conference on Information Security and Privacy (ACISP)*, pages 144–153, July 2002.

[57] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personel Communication Magazine*, 8(4):52–59, August 2001.

[58] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. *In Proceedings of ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[59] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *In Proceedings of The IEEE Aerospace Conference*, pages 1339–1346, March 2003.

[60] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, UCLA, 2003.

[61] The Ohio State University NEST Team. ExScal: Extreme scaling in sensor networks for target detection, classification, tracking, 2004. DARPA, `http://www.cse.ohio-state.edu/exscal`.

[62] D. Tian and N. D. Georganas. A node scheduling schedule for energy conservation in large wireless sensor networks. *Wireless Communications and Mobile Computing Journal*, May 2003.

[63] L. F. W. van Hoesel, T. Nieberg, H. J. Kip, and P. J. M. Havinga. Advantages of a tdma-based, energy-efficient, self-organizing mac protocol for wsns. *IEEE VTC 2004 spring*, May 2004.

[64] H. Wang and Q. Li. Efficient implementation of public key cryptosystems on mote sensors (short paper). *International Conference on Information and Communication Security (ICICS), LNCS 4307*, pages 519–528, December 2006.

[65] L. Wang and S. S. Kulkarni. Proactive reliable bulk data dissemination in sensor networks. *The International Workshop on Distributed Algorithms and Applications for Wireless and Mobiel Systems (DAAWMS)*, November 2005.

[66] L. Wang and S. S. Kulkarni. Gappa: Gossip based multi-channel reprogramming for sensor networks. *In The International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2006.

[67] L. Wang and S. S. Kulkarni. Sacrificing a little coverage can substantially increase network lifetime. *In Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, September 2006.

[68] L. Wang and S. S. Kulkarni. Authentication in reprogramming of sensor networks for mote class adversaries. *The 15th International Workshop on Parallet and Distributed Real-Time Systems (WPDRTS)*, March 2007.

[69] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. *In Proceedings of ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, November 2003.

[70] W. Xiao and D. Starobinski. Poster abstract: Exploiting multi-channel diversity to speed up over-the-air programming of wireless sensor networks. *In Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys) (Poster Session)*, November 2005.

[71] T. Yan, T. He, and J A. Stankovic. Differentiated surveillance for sensor networks. *In Proceedings of the First International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 51–62, November 2003.

[72] F. Ye, G. Zhong, J. Cheng, S. W. Lu, and L. X. Zhang. PEAS: A robust energy conserving protocol for long-lived sensor networks. *In Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[73] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. *In Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1567–1576, June 2002.

[74] Y. Yu, R. Govindan, and D. Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report UCLA/CSD-TR-01-0023, UCLA, May 2001. `http://citeseer.ist.psu.edu/yu01geographical.html`.