MNP: Multihop Network Reprogramming Service for Sensor Networks[1]

Sandeep S. Kulkarni Limin Wang
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

Reprogramming of sensor networks is an important and challenging problem as it is often necessary to reprogram the sensors in place. In this paper, we propose a multihop reprogramming service designed for Mica-2/XSM motes. One of the problems in reprogramming is the issue of message collision. To reduce the problem of collision and hidden terminal problem, we propose a sender selection algorithm that attempts to guarantee that in a 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. We also use pipelining to enable fast data propagation. 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. Finally, we argue that it is possible to tune our service according to the remaining battery level of a sensor, i.e., it can be tuned so that the probability that a sensor is given the responsibility of transmitting the code is proportional to its remaining battery life.*

Keywords: Sensor Networks, Network Reprogramming, Code Dissemination

1 Introduction

Sensor networks have been proposed for a wide variety of application areas. To be practically useful, a sensor network must be able to operate unattended for long periods of time. This requirement introduces several difficulties. First, the environment evolves over time. Predicting the whole set of actions that a sensor node might need to perform is impossible in most applications. Second, 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. Thus, reprogramming sensor nodes, i.e., changing the software running on sensor nodes after deployment, is necessary for sensor networks.

Traditionally, reprogramming is done manually. For example, in Mica-2 motes [1], program code is sent from PC directly to the program memory of a sensor node. Hence, sensor nodes are reprogrammed in a one-by-one fashion. However, as the size of sensor networks continues to grow, this kind of manual reprogramming is no longer feasible. Moreover, even collecting the sensors from the field and reprogramming them using wireless reprogramming (typically in small batches using the single-hop reprogramming (XNP) [2] included in TinyOS 1.0) can be a daunting task. Therefore, reprogramming needs to be performed without physical contact with the sensor nodes.

Network reprogramming in sensor networks poses several new challenges. First, network reprogramming requires 100 percent delivery, which includes two parts: every node in the network must receive the program code, and the code image must be received in its entirety. This is very different from traditional sensor network applications, in which, occasional loss of data is tolerable.

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, delivering the entire program image, of the order of kilobytes over low-bandwidth wireless radio, as required in network reprogramming, requires 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 too many senders are transmitting at the same time. This causes a lot of message collisions, congests the 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 [3]. Reducing the messages sent and received is also important.

Further, memory requirements of network reprogramming should be minimized. Memory is a scarce resource for sensor nodes. For example, only 4k RAM and 128k program memory (ROM) are available in Mica-2 [1] and XSM [4] motes. Because network reprogramming is supposed to be a service resident on every sensor node, high memory usage would limit the available space for normal behavior of other

[1]Email: {sandeep,wanglim1}@cse.msu.edu.
Web: http://www.cse.msu.edu/~{sandeep,wanglim1}.
Tel: +1-517-355-2387, Fax: +1-517-432-1061.
This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF equipment grant EIA-0130724, and a grant from Michigan State University

applications.

With this motivation, in this paper, we present a multihop network reprogramming protocol (MNP) which provides a reliable service to propagate new program code to all sensor nodes in the network over wireless radio. We implement MNP on Mica-2 [1] and XSM [4] motes and simulate it using TOSSIM [5].

Contributions of the paper.

1. We propose a sender selection mechanism, in which source nodes compete with each other based on the number of distinct requests they have received. Through experiments with Mica-2 motes, we show that even a simple greedy approach like this works very well, and has effectively reduced the concurrent sender problem.
2. We use pipelining to enable fast data propagation. Through simulation, we show the effectiveness of pipelining in large scale networks. Moreover, we find that the dynamic behavior reported in [6] (where the propagation speed along the diagonal is significantly less than the speed along the edge) does not exist in MNP.
3. We reduce the active radio time of a sensor node by putting the node into "sleep" state when its neighbors are transmitting. This effectively reduces the idle listening problem and avoids overhearing.
4. We implement MNP in TinyOS Mica-2 and XSM mote platforms, and evaluate its performance through simulation (on TOSSIM) and experiments (on Mica-2 motes).

Organization of the paper. In Section 2, we identify the requirements of the reprogramming problem. In Section 3, we present our code dissemination protocol. We focus on sender selection algorithm, pipelining, and reliability issues. The evaluation results are presented in Section 4. We review related work in section 5, and conclude in Section 6.

2 System Model and Problem Statement

We make no assumptions about the underlying network topology. We require that all sensor nodes receive the exact program image as long as the network is connected. Currently we only consider networks with stationary nodes. We also assume that every node needs to be updated with the same version of code.

In MNP, sensor nodes do not need to have any location information or maintain neighbor status. Sensor nodes make local decisions independently and, hence, the protocol is scalable. We require that a code dissemination protocol meet the following requirements.

1. *Reliability.* This includes both *accuracy* requirement and *coverage* requirement. 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. *Energy efficiency.* The energy used in code dissemination should be low so as to affect the network lifetime minimally.
4. *Low memory usage.* Code dissemination is supposed to be an underlying service running together with other applications. Therefore, the memory and storage requirements should be minimized.
5. *Speed.* New program code should be propagated and installed quickly (i.e., within a few minutes).

Among these requirements, reliability and autonomy are basic and essential requirements for the correctness of code dissemination mechanism. Other requirements are not necessary to ensure correctness, but they are also important and cannot be overlooked for the practical use of any system. Since it is difficult, if not impossible, to fulfill all the design goals in a system, tradeoff has to be made to assure the system's overall functional and performance goals.

3 MNP: Protocol Description

In this section, we present our code dissemination protocol, MNP. We first present our sender selection protocol, which is the core of MNP. We have two versions of the sender selection protocol. In Section 3.1, we first present the basic version of the sender selection protocol. In this version, a node becomes a source node (and starts advertising this fact in its neighborhood) only if it gets the entire new program. This essentially divides a multi-hop forwarding operation into a series of single-hop transmissions. Then, we revise this protocol so that it can be used with pipelining. In Section 3.2, we describe the sender-receiver behavior when a node is forwarding code to its neighbors. In Section 3.3, we discuss the reliability issues, including loss detection and recovery. In Section 3.4, we describe the operation of the protocol as a state machine. In Section 3.5, we discuss the problem when the sensor nodes should reboot with the received program.

3.1 Sender Selection Protocol

In this section, we first present the basic version of our sender selection protocol. Then we revise the protocol so that it can be used with pipelining.

3.1.1 Basic Sender Selection Protocol

In this section, we assume that program is propagated in a hop-by-hop fashion. In each neighborhood, a source node sends program code to multiple receivers. When the receivers get the full program image, they become source nodes, and send the code in their neighborhood. We present our sender selection protocol under this assumption.

Before discussing the protocol in detail, we describe it in the context of an example. Towards this end, consider the example of a sensor network in Figure 1. Suppose A transmits the data object first and nodes B, C, D, E, and G receive this object. 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.

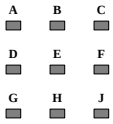


Figure 1. Example sensor network

In our protocol, each source node maintains a variable *ReqCtr* that indicates the number of distinct requests (from different requesters) it has received so far. *ReqCtr* is set to zero when a source starts advertising, and incremented by one every time it receives a *download request* that is *destined* to it from a "new" requester.

Two types of messages are used for sender selection: *advertisement* and *download 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 the advertisement message from a source node, say k, if j needs the new code, then it sends a *download request* to k. The *download request* also contains the value of *ReqCtr* that k sent in the advertisement phase. While the *download 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 *download 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 *download request*, we allow l to be aware of the number of requesters of k. Hence, l can utilize this information to determine who should transmit the code first.

We note that a node sends a *download 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* value to zero, and recalculate its requesters. This is due to the fact that some of old followers of k may have already received the code from l.

After l finishes transmitting the code, it quits the competition temporarily by "sleeping" for a while, so that other sources have better chance to become senders. The purpose of this is to distribute transmission load through the network. When l wakes up and re-enters the "advertise" state, its *ReqCtr* value is reset to zero, and a new round of sender selection starts.

Based on the above discussion, our sender selection protocol can be described as two parts: source part and requester part.

Tasks for Source. In Figure 2, we present the tasks that a source 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 broadcasts an *advertisement* message every random interval (we use random interval to avoid message collision). Every time a source receives a *download request* message, it checks to see if this message is *destined* to it. If it is destined to it, and it is from a "new" requester that the source has never seen before, it increments *ReqCtr* by one. If the *download request* message is destined to some other node and that node has a higher *ReqCtr* value, then it stops advertising and goes to "sleep" state.

If a source node 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 it does, it gives up advertising and goes to "sleep" state. (Note that this cannot cause deadlock, as the node with highest *ReqCtr* - with appropriate tie breaker on node ID - will succeed.) Moreover, the "sleeping" period is proportional to the size of the new program, and lasts for approximately the expected code transmission time. For a "sleeping" sensor node, nothing is active except a timer. When the "sleep" timer fires, the source node wakes up and re-enters "advertise" state.

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

The advertising phase ends when a source node has sent a given number of advertisements continuously (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 advertise with reduced frequency (we exponentially increase the advertise interval if no request is received). Applying different advertise frequencies enables fast data propagation when the network is in active updating state, and saves energy when the network is "stable".

Tasks for Requester. In Figure 3, we show the tasks that a requester performs in the sender selection process. If a node hears an *advertisement* that announces the availability of a new program, it broadcasts a *download request* message, destined to the advertising node. As mentioned earlier, it also puts the *ReqCtr* information of that advertising node in the *download request* message.

3.1.2 Sender Selection with Pipelining

The sender selection algorithm in 3.1.1 is suitable for the case where the program is small or where the network is too small to take advantage of pipelining. For large networks where a large amount of data is sent, pipelining is desirable.To achieve this pipelining, we divide a program into *segments*, each of which contains a fixed number of *packets*. Each segment is assigned a segment ID that is strictly increasing. Nodes must receive the segments sequentially. We make the following changes to the protocol in Figures 2 and 3.

1. Each *advertisement*/*download request* message contains an additional field *SegID* (Segment ID).
2. When a node receives an *advertisement* for a segment that it does not have, it sends a *download request* that

```
Source: (in advertise state)

Broadcast an advertisement message every random interval
After advertising K times (without sleep):
    if (my.ReqCtr > 0)
        Become a sender, and start forwarding code
    else
        Restart advertising, with lower frequency
    endif

During advertise interval:
(a)
if a download 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))
            Stop advertising, 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))
        Stop advertising, go to "sleep" state, my.ReqCtr = 0
    endif
endif

(c)
if "StartDownload" message or data packets arrives
    Stop advertising, go to "sleep" state, my.ReqCtr = 0
endif
```

Figure 2. Tasks of the source in sender selection mechanism

```
Requester:

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

Figure 3. Tasks of the requester in sender selection mechanism

contains the ID of the segment it expects to receive. For example, it the *advertisement* is for segment 3, and the node has received segment 1 in the past, it will ask for segment 2.

3. Whenever a node receives a *download request* for segment y while advertising segment x, if $y < x$, then it starts advertising segment y. This is true even if the *download request* is *not* "destined" to this node.
4. When node l receives an *advertisement* for segment y from node k while it is advertising segment x, if $y < x$, and k has already received at least 1 *download requests*, then node l should go 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 segment.

3.2 Tasks in Downloading a Segment

When a node decides to become a sender, it broadcasts a "StartDownload" message to announce this fact. Then, it starts sending code packet by packet. A node will change to "download" state once it hears a "StartDownload" message with expected segment ID. As a node always receives segments sequentially, the expected segment ID is the highest segment ID the node has received so far plus one. The node in *download* state also 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 matches.

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, there are two choices: the node can go to "fail" state directly; or it can go to the *query/update* phase, during which it requests for the missing packets from its parent. We will discuss the *query/update* phase in the next section.

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 fails. Then it will go to "fail" state.

3.3 Reliability Issues: Loss Detection and Recovery

In MNP, each packet has a unique packet ID. Each receiver is responsible for detecting its own loss. Since the size of the segment is small and pre-determined, we maintain a bitmap (which we call *MissingVector*) of the current segment in memory. Each bit in *MissingVector* corresponds to a packet. All the bits are initially set to 1. When it receives a packet, the corresponding bit in *MissingVector* is set to 0. One additional advantage of this mechanism is that the packets do not necessarily arrive in order. A sensor node can receive packets in any order and from any node.

In MNP, each node has a *ForwardVector*, which is also 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 *download request*, it puts the loss information (its *MissingVector*) in the *download request* message. When the advertising node receives the download request, it marks

its *ForwardVector* according to the loss information. Therefore, an advertising node's *ForwardVector* is the union of the missing packets in the *download request* messages the node has received. A node only sends the packets indicated in the *ForwardVector*. In addition, 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. For the case where larger segments are used, for example, in scenario where pipelining is not expected to be beneficial (small networks), we provide a mechanism to use EEPROM to keep track of lost packets. However, for reason of space, we omit the implementation of that. A reader can find the details in [7].

This algorithm is efficient in that a node will send a packet only if there is some node requesting for it. 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.

We also provide a *query/update* phase, after the sender has finished transmitting the requested packets in the segment. The parent-children relationship also applies in this phase. The sender broadcasts a "query" message to its receivers, which respond by requesting the packets they are missing. A node's request messages are *unicast* to its parent. When a sender receives a request from one of its children, it *broadcasts* the requested packet. The *query/update* phase is optional. This option is desirable in cases where the number of packets lost by the receiver is less than a given threshold.

3.4 The Big Picture

Figure 4 gives an overall picture of MNP. MNP operates as a state machine. Since the *query/update* phase is optional, we have two versions of the state machine, one without *query/update* and one with *query/update*. Both state machines have 6 states in common: *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 go to *fail* state. *Fail* state is a temporary state. A node in *fail* state releases EEPROM resource, and switches to *idle* state immediately. Note that in the state machine with *query/update*, there are additional two states: *query* and *update*, and the associated transitions. A node running MNP is in one of these 8 states.

3.5 When to Reboot

When a sensor node receives all the segments of a program, it can reboot with the new program. Reboot can happen automatically. A node can reboot 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 serve as the source node after reboot. The other choice is to let a node to 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 has received no requests, it assumes that all its neighboring nodes have received the whole program, and then reboots itself with the new program. However, this local estimation of neighbors may be inaccurate, because the messages can be lost or collided, or the neighbors may be in "sleep" state, thus are unable to respond.

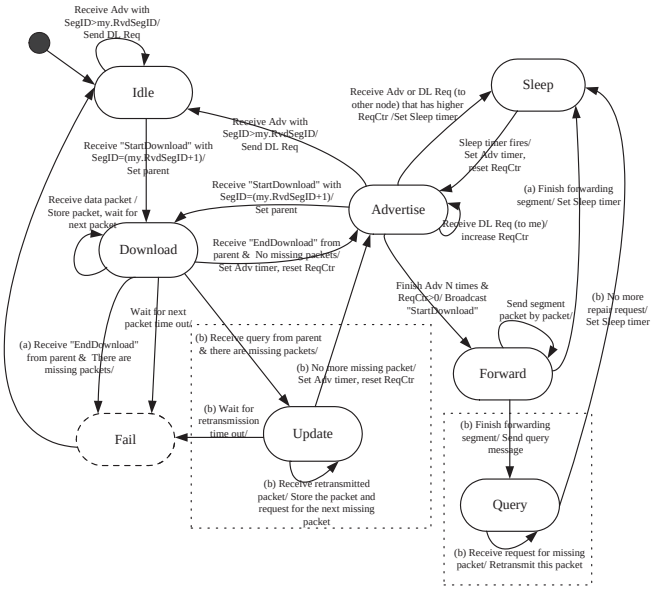


Figure 4. MNP: the state machine. (a) transitions for MNP without *query/update* (b) transitions for MNP with *query/update*

Based on these concerns, we decide not to let sensor nodes reboot automatically. A sensor node will reboot with the new program only when it receives an external "start" signal. The time when the signal is sent should be based on empirical data from experiments. We can also send query messages to individual nodes asking about their status before sending the "start" signal.

4 Evaluation Results

Our target platform is TinyOS Mica-2/XSM motes, with 433MHz radio. A Mica-2/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 Mica-2 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, Mica-2 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 [5]. 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 Mica-2 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.

4.1 Experiments with Mica-2 Motes

TinyOS allows developers to specify the power level a Mica-2 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.

4.1.1 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 5(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 5(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 5(a), most of the sensors receive code directly from the base station. When we reduce power level to 1, as shown in Figure 5(b), more sensors are not covered by the base station, thus have to obtain code from other intermediate nodes.

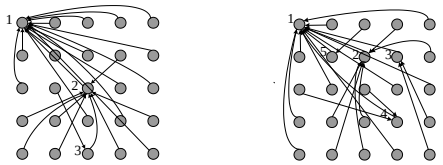


Figure 5. Indoor experiments for 5 by 5 grid with (a) power level = 2, time = 5 minutes; (b) power level = 1, time = 8 minutes. Program size: 1500 packets (33KB).

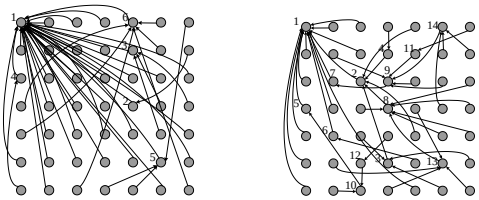


Figure 6. Outdoor experiments for 7 by 7 grid with (a) full power level, time = 25 minutes; (b) power level = 10, time = 35 minutes. Program size: 1500 packets (33KB).

4.1.2 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 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 6 shows the parent-children relationship and the order of source nodes becoming senders for 7 by 7 grid. Figure 7 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 5, 6 and 7, 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

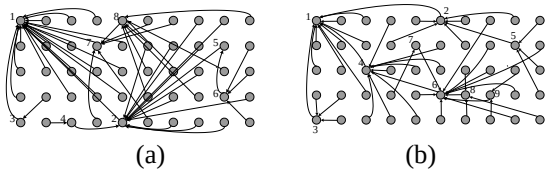


Figure 7. Outdoor experiments for 5 by 10 grid with (a) full power level, time = 35 minutes; (b) power level = 10, time = 45 minutes. Program size: 1500 packets (33KB).

Table 1. Power 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 Data	1.111
EEPROM Write Data	83.333

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 similar. 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).

4.2 Simulation Results

In TOSSIM, the network is modelled as a directed graph. Each vertex in the graph is a node. Each edge has a bit error probability, representing the probability 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, based on empirical loss data gathered from real world. Since TOSSIM does not capture energy consumption, we calculate the energy consumption by counting the operations performed during reprogramming. In Table 1, we list the costs of various operations from [3].

We note that the energy consumed in idle listening is comparable to the energy consumed in transmitting/receiving, and it is proportional to the active radio time. Reprogramming typically lasts from several minutes to several hours. If a node keeps its radio on at all time, the vast majority of energy is wasted in idle-listening. In MNP, a node turns off its radio when it loses in the sender selection algorithm or one of its neighbors is transmitting a segment in which it is not interested. In the following discussion, we count the active radio time, as well as the completion time. The active radio time is even more important than the completion time, because it decides the amount of energy that a node actually consumes.

The number of messages sent and received is also an important metric for energy consumption. We use sender selection to address the message collision problem and try to select the senders that tend to have the maximal effect. This effectively reduces the number of transmissions and receptions. Moreover, by including the loss information in the *download request* messages, we further reduce the message transmission by letting a sender send only the packets that are requested by its neighbors. Finally, regarding EEPROM writes, we guarantee that each packet is written to EEPROM only once.

In this section, we first measure the active radio time of nodes for propagating a given size program in a square topology. Then we examine the number of transmissions and receptions based on location and time. Finally, we show the code propagation progress. In the current implementation, each segment has 128 data packets and each data packet has 22 bytes data payload. The sensor nodes are deployed in a grid topology. The distance between every two nodes is kept constant at 10 feet apart. We assume that initially only the base station, the node at the bottom-left corner, has the new program.

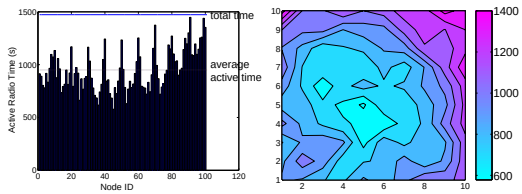


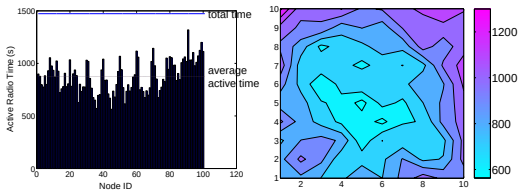
Figure 8. Active radio time of nodes in a 10 by 10 network. Program size: 14KB (5 segments). Completion time: 24 minutes. Average active radio time: 949 seconds.

4.2.1 Active Radio Time and Completion Time

In Figure 8, we show the active radio time distribution in a 10 by 10 network. The simulation starts by the base station sending a 5 segment program (14KB). It takes approximately 24 minutes for the program to go through the whole network. The active radio time for an average node is 949 seconds. In this case, we save about 38% of energy spent on idle listening by turning off radio. Figure 8 (a) shows the active radio time of each node. Figure 8 (b) shows the same values based on location of sensors. As shown in Figure 8, the active radio time of a node in the network is closely related to its location. The active radio time for the nodes in the center is approximately half (or even less) of those on the edges. Note that this pattern is similar to the reception distribution, which we show in Figure 11 (b). The nodes in the center receive many more messages than the ones on the edge or at the corner, thus they get the code and become source nodes earlier. Since there is only one sender at a time in any neighborhood, after a node has got the code, it spends most of the time in "sleeping" state, during which the radio is off. On the other hand, a node at the corner receives fewer messages than the center nodes, and has to spend more time trying to get the code, during which the radio is always on.

We also notice that the nodes that are far away from the base station keep their radio on most of the time during reprogramming. Initially, the radio is on. The nodes close to the base station get the code immediately, while the nodes that are far away from the base station have to wait until the "propagation wave" arrives, and the radio is always on when they are waiting. To solve this problem, we can use a protocol such as S-MAC [8] or SS-TDMA [9] that allows a node to synchronize its wake up and sleep time with its neighbors. In this case, a node could sleep for most of the time before the propagation wave arrives. Hence, in Figure 9, we show the active radio time after the node has received its first advertisement message. As we can see, in this case, the active radio time of all nodes is closer to each other than that in Figure 8.

In Figure 10, we show the completion time, the active radio time, and the active radio time after the sensor has received its first advertisement message, for various program size, from 1



(a) (b)

Figure 9. Active radio time of nodes without initial idle listening in a 10 by 10 network. Program size: 14KB (5 segments). Completion time: 24 minutes. Average active radio time: 862 seconds.

segment (2.8KB) to 10 segments (28.2KB). As we can see, the completion time is linear with the program size, and the active radio time is around 60% of the completion time.

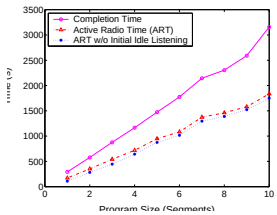
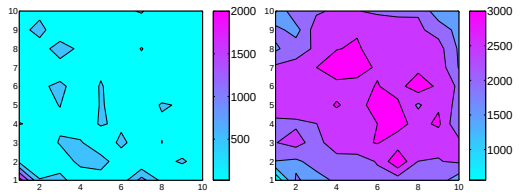


Figure 10. Completion time, active radio time and active radio time without initial idle listening for various program size in a 10 by 10 network. Program size is from 1 segment (2.8KB) to 10 segments (28.2KB).

4.2.2 Transmissions and Receptions

In Figure 11, we show the transmission and reception distribution in a 10 by 10 network. The size of the program is 14KB (5 segments). We note that the number of messages sent by each node is low, on average 400 messages, including all data and control (advertisements, requests, etc.) messages. The node sending the most number of messages is the base station, since all the data messages are originated from it. The nodes that are close to the base station get code earlier than those that are far away from the base station. Thus they become source nodes earlier and send more data packets. In the reception distribution, the nodes in the center receive many more messages than the ones on the edge or at the corner. This is due to the fact that a node in the center has more neighbors than that at the corner.

In Figure 12, we show different types of messages transmitted in the network in a one-minute window. The number of



(a) (b)

Figure 11. Transmission and reception distribution in a 10 by 10 network. (a) messages transmitted (b) messages received. Program size: 14KB (5 segments).

data messages transmitted remains almost constant during the entire process, indicating a smooth data propagation flow.

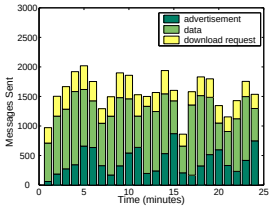


Figure 12. Overall advertisements, download requests, and data messages transmitted in a one-minute window in a 10 by 10 network. Program size: 14KB (5 segments).

4.2.3 Propagation Progress

Figure 13 shows the code propagation progress when we send one segment (2.8KB) in a 14 by 14 network. The distance between every two nodes is still 10 feet. As we can see, data is propagated at a fairly constant rate from the base station to the other end of the network.

5 Related Work

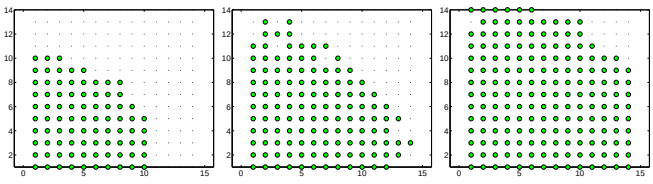
In this section, we discuss work in the areas of network reprogramming and suppression schemes.

Network reprogramming. The work on network reprogramming/data dissemination can be classified into two types: entire code image delivery ([2, 6, 10, 11]) and difference-based application adjustment ([12, 13]). Entire code delivery provides the basic reprogramming functionality, while difference-based approach can be used to send differences between versions. MNP follows the former approach. However, we note that our solution is complementary to difference-based approaches. In other words, our sender selection and loss recovery approaches can be used to improve difference-based approaches as well.

The existing work on network reprogramming includes TinyOS single-hop network reprogramming (XNP) [2], MOAP (Multihop Over-the-Air Programming) [10], and Deluge [6]. All of them are entire code delivery approaches, and are built on TinyOS platform.

TinyOS has included single-hop network reprogramming support (XNP) for Mica-2 motes since the release of version 1.0. In XNP, one source node (the base station) broadcasts the code image to all the nodes within its radio range.

MOAP is a multihop network reprogramming approach. MOAP disseminates code in a hop-by-hop fashion, that is,



(a) (b) (c)

Figure 13. Code propagation progress for sending one segment (2.8KB) in a 14 by 14 network (a) 30% of time (b) 60% of time (c) 90% of time

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.

Deluge is another multihop network reprogramming approach. MNP shares many ideas with Deluge, such as advertise-request-data handshaking (based on SPIN [14]), dividing a code image into equally sized *pages*, pipelining the transfer of pages, and using a bit vector to keep track of loss within a page. In contrast to MNP, Deluge (as well as XNP and MOAP) requires that radio is always on during reprogramming. Therefore a node's idle listening time is the same as the completion time. Since the energy spent on idle listening is the major source of energy consumption, we compare the average active radio time in MNP with the completion time in Deluge. As reported in [6], for Deluge, the time for sending 240 packets (5KB) through a 10 by 10 network is more than 700 seconds. In MNP, it takes about 24 minutes to send 640 packets (14KB) through a 10 by 10 network. The active radio time in average is about 930 seconds (or 862 seconds without the initial idle listening). Therefore we send almost 3 times the data using 30% more time. If we consider a program of similar size, that is, sending 2 segments (256 packets, 5.6KB) through a 10 by 10 network. It takes 577 seconds to complete. The active radio time is only 352 seconds (or 273 seconds without the initial idle listening). Therefore, MNP saves energy by turning off a node's radio when it is not supposed to transmit or receive.

In [6], the dynamic behavior of Deluge is investigated. It is found that when the network is dense, the propagation speed along the diagonal is significantly less than the speed along the edge. One of the main causes of this behavior is the hidden terminal problem, which occurs when two senders out of range of each other transmit packets to the same receiver at the same time, thus causing collisions at the receiver. In MNP, we solve the message collision problem using the sender selection protocol. And we address the hidden terminal problem by including "requester counter" information of a source node in *download request* messages. Hence, we did not observe this kind of behavior (as shown in Figure 13).

To address the very resource constrained nature of sensor nodes, Maté [12], and its successor, Bombilla, is included in TinyOS. Bombilla 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, Bombilla allows new programs to be forwarded and installed quickly through a network. However, there is severe limitation on the sort of applications that can be built.

The approaches we mentioned so far use CSMA-based MAC protocol. We can also build up a reprogramming service based on TDMA (e.g., [8, 9]). 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 only applicable when the network topology is well defined (e.g., a grid).

Suppression schemes. *Message implosion* or *broadcast storm* problem [15] 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 [16], and broadly used in almost all flat structured or cluster-based protocols, such as LEACH [17], SINA [18].

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) [19] and Trickle [20].

Our sender selection protocol is also delay based. We use "number of requesters" as the criteria to choose sender. Our goal is to find the "good" senders who have many "followers".

6 Conclusion and Future Work

In this paper, we presented a multihop network reprogramming protocol, MNP, that is targeted at Mica-2/XSM motes. MNP uses a sender selection protocol to reduce message collision and to address the hidden terminal problem. When multiple sensor nodes compete for being the potential sender, the sender selection algorithm attempts to find a node whose transmission of the program code is likely to have the most impact. Based on the experiments presented in Section 4, our protocol ensures that at a time at most one sender is active in any neighborhood. Also, MNP propagates the code in a pipelined fashion.

We evaluated MNP through experiments on Mica-2 motes and simulation on TOSSIM. In our experiments and simulation, we kept the base station at the corner. Hence, we expect that this algorithm can be easily extended to the case where the network size is 4 times larger (twice the length and breadth) and the base station is in the center. 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 [21]. In the first demonstration, we deployed 50 Mica-2 sensors on a grass field and reprogrammed all the sensors with Lites code [22]. In the second demonstration, we showed that MNP scaled well in a larger network of 100 XSM sensors.

In MNP, some nodes are selected to transmit the code whereas others can "sleep" to save power and to prevent interference. This effectively reduces the idle listening problem. We showed that this approach helps us in significantly reduc-

ing the energy usage. 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 followers and, hence, it will lose in the sender selection. It follows that with this modification, the probability that sensor forwards the code to others depends on its remaining battery level. With this modification, the responsibility of transmitting the code will be evenly divided among the sensors.

Our simulation focused on the energy consumption during reprogramming process, of which idle listening and messages transmissions and receptions are the two major sources. We showed that by turning off a node's radio when it is not transmitting or receiving, we can greatly reduce the idle listening time. Further, because our sender selection protocol reduces message collisions as well as the hidden terminal problem, we did not observe the dynamic behavior (reported in [6]) where the propagation speed in the center is less than that along the edge.

Nodes running MNP are put into "sleep" state occasionally and wake up when the sleeping timer fires. Deciding the sleeping period is a trade off. If a node wakes up frequently, a lot of energy is wasted on idle listening and turning on and off radio. But if a node sleeps for too long time, it may miss the advertisements sent by its neighbors. One promising option is to combine MNP with time scheduling mechanisms such as TDMA, so that each node can sleep and wake up at predefined time slots, and a node will send the advertisements only when its neighbors are awake.

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.

References

[1] J. Hill and D. Culler. Mica: A wirleess platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

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

[3] 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.

[4] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. *In Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN), Special track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005, to appear.

[5] 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.

[6] 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, to appear.

[7] S. S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. Technical Report MSU-CSE-04-19, Department of Computer Science, Michigan State University, May 2004.

[8] 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.

[9] S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In *Sensor Network Operations*. IEEE Press, 2004, to appear.

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

[11] S-J. Park, R. Vedantham, R. Sivakumar, and I. F. Akyildiz. A scalable approach for reliable downstream data delivery in wireless sensor networks. *In Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 78–89, May 2004.

[12] 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.

[13] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *the 2nd ACM international conference on Wireless sensor netowrks and applications*, September 2003.

[14] J. Kulik, W. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. volume 8, pages 169–185, 2002.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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`.

[22] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. 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.