Available online at www.sciencedirect.com
ScienceDirect
Ad Hoc Networks 7 (2009) 42–62
Ad Hoc Networks
www.elsevier.com/locate/adhoc

Efficient wireless reprogramming through reduced bandwidth usage and opportunistic sleeping

Rajesh Krishna Panta [a,*], Saurabh Bagchi [a], Issa M. Khalil [b]

[a] *Dependable Computing Systems Lab, School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN 47907, United States*
[b] *College of Information Technology, United Arab Emirates University, United Arab Emirates*

Received 30 July 2007; received in revised form 12 November 2007; accepted 27 November 2007
Available online 15 December 2007

Abstract

Wireless reprogramming of a sensor network is useful for uploading new code or for changing the functionality of existing code. Reprogramming may be done multiple times during a node's lifetime and therefore a node has to remain receptive to future code updates. Existing reprogramming protocols, including Deluge, achieve this by bundling the reprogramming protocol and the application as one code image which is transferred through the network. The reprogramming protocol being complex, the overall size of the program image that needs to be transferred over the wireless medium increases, thereby increasing the time and energy required for reprogramming a network. We present a protocol called Stream that significantly reduces this bloat by using the facility of having multiple code images on the node. It pre-installs the reprogramming protocol as one image and equips the application program with the ability to listen to new code updates and switch to this image. For a sample application, the increase in size of the application image is 1 page (48 packets of 36 bytes each) for Stream and 11 pages for Deluge. Additionally, we design an opportunistic sleeping scheme whereby nodes can sleep during the period when reprogramming has been initiated but has not yet reached the neighborhood of the node. The savings become significant for large networks and for frequent reprogramming. We implement Stream on Mica2 motes and conduct testbed and simulation experiments to compare delay and energy consumption for different network sizes with respect to the state-of-the-art Deluge protocol.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Network reprogramming; Sensor networks; Deluge; Three way handshake; TOSSIM

1. Introduction

Large scale sensor networks may be deployed for long periods of time during which the requirements from the network or the environment in which the nodes are deployed may change. The change may necessitate uploading a new code or retasking the existing code with different sets of parameters. We will use the term *code upload* for referring to both these forms. A primary requirement is that the reprogramming be done while the nodes are *in situ*, embedded in their sensing environment. This has spurred interest in remote multihop reprogramming protocols over the wireless link. For such

* Corresponding author. Tel.: +1 7654099988.
E-mail addresses: rpanta@purdue.edu (R.K. Panta), sbagchi@purdue.edu (S. Bagchi), Ikhalil@uaeu.ac.ae (I.M. Khalil).

1570-8705/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.adhoc.2007.11.015

reprogramming, it is essential that the code update be 100% reliable and reach all the nodes that it is destined for. The code upload should be fast since the network's functionality is likely degraded, if not reduced to zero, during the reprogramming period. It is also important to minimize the resource cost of the reprogramming.

It is conceivable that the process of code upload will be infrequent for many deployments and therefore it may appear that its resource consumption need not be optimized. However, consider that the sensor network environment has inherent unreliability in the wireless links that may have transient failures. Thus the environment is dynamic with nodes coming in and out of periods of disconnectedness. Also, the network may have nodes added after the initial deployment while new code may be injected at arbitrary points in time. Since in most deployments, the sensor network is expected to operate over extended periods of time, it is possible that the parameters for the application, such as the monitoring period, change, necessitating retasking. The code dissemination therefore cannot be considered a one shot process and it becomes important to minimize the resource consumption used in network reprogramming. Also, the resource cost which is incurred during the quiescent or steady state of the network[1] must be optimized since that is the dominant phase in the network lifetime.

A few researchers have proposed protocols for reprogramming in sensor networks, the state-of-the-art being defined by three protocols – Deluge [6], MNP [7], and Freshet [8]. Common to the three protocols is the notion of transferring the code image in chunks of pages on a hop-by-hop basis with each node disseminating code to its immediate neighbor through a three-way handshake of advertisement, request, and actual code transfer. MNP and Freshet build on Deluge and respectively optimize the transfer through judicious sender selection for dense networks and sleep-awake protocols for large networks.

The critical problem that besets all three protocols is *what* is transferred. Common intuition would be to transfer just what is needed, in other words, the application image (or the image of the updates to the application). However, each protocol transfers the image of the entire reprogramming protocol *together with* the minimally necessary part. Since the reprogramming protocols are of considerable complexity, the inflation in the program image size[2] that gets transferred over the wireless medium increases greatly. The exact amount of increase is application specific – for a simple stand-alone application of 1 page, the increase is 20 folds, while for a communicating application of the same size, the increase is 11 folds. In a sensor network environment, this is problematic. First, the network links are prone to transient failures and yet, the code upload process needs to be 100% reliable. Second, the networks are envisaged to be large and the cost of larger image is incurred at every hop and does not get amortized. Third, it puts pressure on multiple scarce resources of a node – communication bandwidth leading to communication contention, and program Flash memory. The authors of Deluge argue convincingly that it is difficult to improve over Deluge the rate of transfer over the wireless link. Therefore, the logical approach appears to be to optimize *what* needs to be transferred, keeping the basic mode of transfer the same as in Deluge.

This thinking gives rise to our protocol called *Stream*, which was introduced by us in [17]. Stream transfers close to the minimally required image size by segmenting the program image into an application image and the reprogramming protocol image. It transfers over the wireless link the former with a minimal addition. It pre-installs in each node, before deployment, the reprogramming protocol image. Stream utilizes the ability to segment the external Flash memory into multiple images and stores the two in two different image areas. An application is modified by linking it to a small component called *StreamApplicationSupport* (Stream-AS) while *StreamReprogrammingSupport* (Stream-RS) is pre-installed in each node. Stream-AS is generic and can be inserted in any TinyOS application through the insertion of two lines of nesC code. Stream-RS builds on Deluge to operate in the changed mode. Overall, Stream's design principle is to limit the size of Stream-AS and providing it the facility to switch to Stream-RS when triggered by a code update

[1] Quiescent does not mean the node is idle. It means there is no activity related to code upload, but the node is running its application and doing its normal activity, such as monitoring.

[2] We use the term *application image* to refer to the user application that needs to run on the node, *reprogramming protocol image* to refer to the protocol components for protocols, such as Deluge, MNP, or Freshet, and *program image* to the combined image that gets transferred over the wireless medium.

related message. The advantage afforded by Stream is demonstrated over Deluge, though it can apply to any of the three protocols, since the problem Stream addresses is shared by each. What would change in applying to a different protocol is that Stream-RS will be based on that protocol.

There are several challenges to implementing the basic idea of Stream in the mote platform. First, the node that has been updated with the recent code needs to remain receptive to future code updates. Thus, it cannot be running just the application. The mote platform does not support multi-tasking and therefore the two programs (reprogramming protocol and application) cannot be executing concurrently. A design option we explored was to pre-install the reprogramming protocol components in the node and dynamically link it to the application to create a single executable image once the application is uploaded. However, TinyOS does not provide a linking facility on the node itself.[3] Second, it is unreasonable to assume that the code update will always occur according to a preset schedule in which case the node could have queried the base station for it. Third, Stream has to consider the possibility that new nodes may be introduced into the network and may query a given node for coming up-to-date with the latest version of the code. Thus a node cannot be content to handle just its own need for staying up-to-date.

When a node has received all its code update, Stream optimizes the steady-state energy expenditure by switching from a push-based mechanism (where the node periodically sends advertisements) to a pull-based mechanism where a newly inserted node requests for the code. The benefit of Stream shows up in fewer number of bytes transferred over the wireless medium leading to increased energy savings and reduced delay for reprogramming. To further reduce the energy used in reprogramming the sensor network, Stream causes the nodes to be involved in reprogramming only when they are actually required to do so, i.e., a node is neither woken up nor switched over from its application duties till the new code has reached the neighborhood and the node has to be involved in the three way handshake for getting the code. Freshet [8] reduces the reprogramming energy by cleverly estimating how long a node can sleep before the new code, after being injected at one point in the network, arrives its vicinity. However, due to the variability of the wireless channel, the estimate made by Freshet based on the hop count is often inaccurate. An inaccurate estimate either causes higher energy expenditure (if the time estimate is too low) or higher delay in completing the reprogramming (if the estimate is too high). Stream achieves the goal without needing to estimate the time, but by rebooting the node from Stream-RS for the purpose of reprogramming only when the new code arrives at one of its neighbors. As a result, the user application running on the node can put the node to sleep till the time to reboot comes. This opportunistic sleeping feature of Stream is useful in conserving the energy in resource constrained sensor networks, especially for large networks where the amount of time to disseminate the code can be quite significant (tens of minutes). Coupled with this fact is the observation that reprogramming is not a one-time task, but rather is done periodically, and quite frequently, in some networks. Note that reprogramming includes the task of updating some parameters in the existing code, which may be more frequent than uploading a new version of the code itself.

We demonstrate the above mentioned claims by implementing Stream in nesC for the Mica2 mote platform. We conduct experiments with Deluge and Stream on a real small-sized testbed (of up to 16 nodes) in linear and grid topologies. The output metrics we measure are number of bytes transferred (which relates to the energy spent) and the delay. We see that Deluge requires 63–98% more reprogramming time and transfers 75–132% more number of bytes than Stream for the grid topologies. To evaluate Stream for larger sized networks, we use the TOSSIM simulation environment. We present a mathematical analysis to evaluate the performance of Stream and compare it to the ideal case when exactly the application image is transferred. The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 provides the detailed design of Stream. Section 4 presents the mathematical analysis. Section 5 explains the testbed experimental setup, and presents the results, details the simulation setup and provides the simulation results. Section 6 discusses the opportunistic sleeping mode of Stream. Section 7 provides a brief explanation of how user application's sleep/awake scheme affects reprogramming and Section 8 concludes the paper.

[3] Interestingly, these constraints are also found in other common sensor node platforms, such as, Sensoria's WINS and JPL's node.

2. Related work

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

The push–pull method for data dissemination through the three way handshake of advertisement-request-code has been used previously in sensor networks with sensed data taking the place of code. Protocols such as SPIN [2] and SPMS [3] rely on the advertisement and the request packets being much smaller than the data packets and the redundancy in the network deployments which make several nodes disinterested in any given advertisement. However, in the data dissemination protocols, there is only suppression of the requests and the data sizes are much smaller than the entire binary code images.

The earliest network reprogramming protocol XNP [4] only operated over a single hop and did not provide incremental updates of the code image. The Multihop Over the Air Programming (MOAP) protocol extended this to operate over multiple hops [5]. MOAP introduced several concepts which are used by later protocols, namely, local recovery using unicast NACKs and broadcast of the code, and sliding window based protocol for receiving parts of the code image. However, MOAP did not leverage the pipelining effect with segments of the code image.

The three protocols that are substantially more sophisticated than the rest and define the state-of-the-art today are Deluge, MNP, and Freshet. All use the three way handshake for locally propagating the code. Deluge [6] was the earliest and laid down some design principles used by the other two. It uses a monotonically increasing version number, segments the binary code image into pages, and pipelines the different pages across the network. It builds on top of Trickle [14], a protocol for a node to determine when to propagate code in a one hop case. The code distribution functions through a three-way handshake protocol of advertisement, request, and broadcast code. The operation of each node is periodic according to a fixed size time window. The first part of the window is for listening to advertisements and requests and sending advertisements. The second part of the window is for transmitting or receiving code corresponding to the received requests. Within the first part of the time window, a node randomly selects a time at which to send an advertisement with meta-data containing the version number, the number of complete pages it has, and the total number of pages in the image of this version. When the time to transmit the advertisement comes, the node sees whether it has heard a threshold number of advertisements with identical meta-data, and if so, it suppresses the advertisement. When a node hears code that is newer than its own, it sends a request for that code and the lowest number page it needs, to the node that advertised the new code. In the second part of the periodic window, the node transmits packets with the code image, corresponding to the pages for which it received requests. A receiving node only fills its pages in monotonically increasing order thereby eliminating the need for maintaining large state for missing holes in the code. For receiving the code, each node uses the shared broadcast medium that allows overhearing and can fill in a page requested by a neighbor.

The design goal of MNP [7] is to choose a local source of the code which can satisfy the maximum number of nodes. They provide energy savings by turning off the radio of non-sender nodes. Freshet [8] is different in aggressively optimizing the energy consumption for reprogramming. It introduces a new phase called blitzkrieg when the code update is started from the base node. During the blitzkrieg phase, information about the code and topology (primarily the number of hops a node is away from the wave front where the code is at) propagates through the network rapidly. Using the topology information each node estimates when the code will arrive in its vicinity and the three way handshake will be initiated – *the distribution* phase. Each node can go to sleep in between the blitzkrieg phase and the distribution phase thereby saving energy. Freshet also optimizes the energy consumption by exponentially reducing the meta-data rate during conditions of stability in the network when no new code is being introduced, called the *quiescent phase*.

The reprogramming protocols discussed above transmit the complete code image during reprogram-

ming. In addition to these full image replacement algorithms, there are some techniques proposed in the literature which reduce the number of packets transmitted during reprogramming by comparing the new code with the previously installed software and transmitting only the difference. Using a diff-like approach, Reijers and Langendoen [19] compute the *diff script* which captures the difference between the old and the new code and consists of copy, insert, address repair and address patch operations. This approach reduces the network traffic but the drawback of their approach is that the address patching is dependent on the regular structure of the instruction set architecture and the authors demonstrate the protocol for a specific customized node called EYES. Also, their work is focused on the encoding scheme and does not demonstrate a fully functional implementation of the reprogramming. Jeong and Culler [20] describe an incremental network reprogramming protocol which uses the Rsync algorithm [18] to find the blocks of the code that are identical in the new and the old code images. A drawback of their approach is that their protocol is meant for single-hop reprogramming. Also, since they do not use the knowledge about the application structure, even small code shifts can result in a large number of address patches leading to high bandwidth consumption. Koshy and Pandey [21] describe a scheme that uses incremental linking on the base node. Their work is also focused on generating the difference between the two code images and does not implement a fully functional reprogramming protocol. FlexCup [24] uses run time linking on the sensor nodes. It is implemented on the Mica2 emulator called ATEMU [22] and not on any real hardware. Further since the compiled components are stored in the external flash, they require more space than Deluge or Stream. Also, the size of the reprogramming protocol is significantly larger than Deluge or Stream. Contiki [23] is an operating system developed for resource constrained systems like sensor networks. Based on Contiki, Dunkels et al. [25] describe a reprogramming approach using dynamic linking on the sensor nodes. This approach is tied to Contiki's structure of differentiated application and OS components and seems difficult to apply to TinyOS. Applications such as Tiny Diffusion [10], Maté [11], and TinyDB [12], use concise, high-level virtual code representations to give programs that are 20–400 bytes long. Unlike these incremental approaches, Stream uses the full image replacement technique and still manages to greatly reduce the amount of data transmitted during reprogramming. It can be integrated with an incremental reprogramming approach whereby instead of sending the entire application as Stream-AS, only the difference can be sent.

3. Stream design

3.1. Design approach

Stream builds on the code distribution method of Deluge. It optimizes the number of bytes that needs to be disseminated over the wireless medium so that instead of transferring the entire Deluge component along with the new application, only a small subset of reprogramming functionality is included in the program image. The idea is to have all nodes in the network be pre-installed with the `Stream-ReprogrammingSupport` (Stream-RS) (Fig. 1) component that includes the complete functionality for network reprogramming. Stream-RS is installed as image 0. The application image augmented with the `Stream-ApplicationSupport` (Stream-AS) component that provides minimal support for network reprogramming is installed as image 1. The addition to the size of the program image over the application image size with Stream is significantly less than in the Deluge case. When a new program image is to be injected into the network, all the nodes in the network running image 1 reboot from image 0 and the new image is injected into the network using Stream-RS. The new image again includes Stream-AS and we avoid the entire Deluge component from being transferred to all the nodes each time the network needs to be reprogrammed. This modification does not entail modification of the application on the part of the user. Instead of adding the Deluge component, she adds the much smaller component (Stream-AS) to her application. Both are localized two line changes in the application code.

The saving in terms of the number of pages transferred is quite significant. The exact figure depends on the application. Any application that uses radio

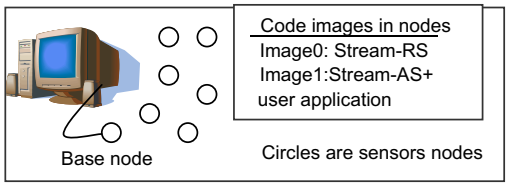

Fig. 1. Images in Stream.

communication will need to add about 11 more pages if Deluge is used while Stream-AS adds only one more page. We stress that this benefit is demonstrated here for Deluge, but applies equally to all the current network reprogramming protocols since each transfers the entire protocol image along with the application image.

3.2. Protocol description

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

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

1. In response to the reboot command from the user, all nodes in the network reboot from image 0. This is accomplished as follows:
   a. The base node executing image 1 initiates the process by generating a command to reboot from image 0. It broadcasts the reboot command to its one hop neighbors and itself reboots from image 0.
   b. When a node running the user application receives the reboot command, it rebroadcasts the reboot command and itself reboots from image 0.
2. Once the reboot command reaches all nodes, all nodes start running Stream-RS. Then the new user application is injected into the network using Stream-RS.
3. Stream-RS starts to reprogram the entire network. It does so by using the three way handshake method where each node broadcasts the advertisement about the code pages that it has. When a node hears the advertisement of newer data than it currently has, it sends a request to the node advertising newer data. Then the advertising node broadcasts the requested data. Each node maintains a set S containing the node ids of the nodes from which it has received the requests.
4. Once the node downloads the new user application completely, it performs a single-hop broadcast of an ACK indicating it has completed downloading.
5. Upon receiving the ACK from a node, it removes the id of that node from the set S.
6. When the set S is empty and all the images are complete (by complete we mean that all pages of all images have been downloaded), the node reboots from image 1. So, after sometime the entire network is reprogrammed and all nodes reboot from image 1.

3.3. Handling incremental network deployment

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

3.4. Design of Stream-AS

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

Stream-AS provides functionality to reboot from image 0 when new nodes are introduced to the network. When new nodes join the network, they periodically broadcast the advertisement. After one-hop neighbors of these new nodes hear the advertisement, they reboot from image 0 (Stream-RS). Then Stream-RS takes care of sending the new application image to the new nodes. One disadvantage of the current implementation of Stream is that the new nodes must be running image 0 so that upon hearing the advertisement from these nodes, already deployed nodes can reboot from Stream-RS.

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

```
Components DelugeC;
Main.StdControl → DelugeC;
```

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

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

3.4.1. Steady-state behavior

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

3.5. Design of Stream-RS

Stream-RS, preinstalled in all nodes as image 0 and executed only during reprogramming phase, is responsible for actual image transfer among the nodes in the network. It is based on Deluge with the significant changes mentioned below. When new application image is to be injected into the network, all nodes reboot from Stream-RS. Then, reprogramming is done by using a three-way handshake (Fig. 2) in which each node broadcasts the advertisement about the code pages that it currently has. A node, upon hearing the advertisement of newer data than it currently has, sends a request to the node advertising newer data. The advertising node then broadcasts the requested code pages. Deluge optimizes this reprogramming method by proper choice of the time when advertisements and code pages are sent. For a complete discussion of Deluge, see [6].

3.5.1. Changes from Deluge

Once reprogramming is done we want all the nodes to reboot from the new application automatically. One obvious approach would be to reboot each node from the user application after it completes downloading the new application. But the flaw with this approach is that even though a node has completed downloading the new application, it may still be serving other nodes in the network. Therefore the node needs to continue to run Stream-RS. When a node receives a request for data, it puts the node-id of the requesting node in the set S. This set S is not shared and is maintained by each node. This structure is essential because a design philosophy of Stream is to have all nodes running the user application all the time except

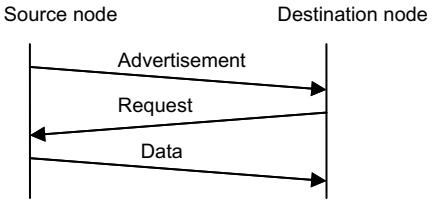

Fig. 2. Three-way handshake for data dissemination.

when reprogramming is being done. Without knowledge of the nodes to which node A is currently sending code, A may reboot from image 1 after it has downloaded all the pages of the new user application even though some nodes in the network may still be receiving code fragments from A.

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

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

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

$$A \cdot S = \phi \wedge A \cdot \#\mathrm{pages} = \mathrm{Total\ number\ of\ pages}.$$

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

There is a subtle drawback to the synchronization in Stream-RS. A node A may be serving a node B without B having explicitly sent a request to A. Thus node B would never be included in A's set S. Let us consider a scenario when a node n_1 hears the advertisement of newer data than it currently has from node n_2 during the reprogramming phase. Before node n_1 sends request for the new data, some other one-hop neighbor n_3 of the advertising node n_2 may send the request. In response to the request from node n_3, n_2 broadcasts the code. So, the node n_1 may never send the request to node n_2 but keep on receiving the code from node n_2, triggered by request from node n_3. If all the nodes that explicitly request data from the advertising node n_2 complete downloading the new application earlier than the node n_1, node n_2 will reboot from the new application. This leaves node n_1 in the middle of downloading the new application. This drawback, however, does not pose a correctness problem, but a performance problem. This is due to the design that after the advertising node n_2 has rebooted from the user application, it still can hear the advertisement sent by the node n_1 due to Stream-AS. Upon hearing the advertisement, node n_2 will reboot from Stream-RS and start sending code to node n_1 through the three-way handshake.

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

4. Stream analysis

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

4.1. Reprogramming time

In this section, we analyze the reprogramming time for a grid network. We assume that the transmission range is equal to $\sqrt{2}$ times the grid spacing. The reprogramming model that we use for the analysis is an approximation of the behavior of Stream. In it, we divide the time line into fixed-size rounds. The source sends the advertisement at the beginning of each round and the destination, the one hop neighbor of the source that hears the advertisement, sends one request for each new advertisement received. We assume, for tractability of analysis, that the advertisement and the request packets are reliably delivered. This can be achieved in practice by either having a separate control channel or by transmitting the control signals multiple times to give a desired reliability. If this assumption is not true, then the multi-hop reprogramming time we find is a lower bound rather than the exact time. Once the source receives the request, the data packets are sent immediately. If not all the data packets in a page can be sent to the destination, the remaining data packets are sent over the following one or more rounds. The time T_r is defined as the time to send a new advertisement, receive a request, and send all the N_{pkt} packets of the page being advertised when the link reliability is 1.0. The number of rounds that it takes for all the packets in a page

to be received at the destination is thus a random variable. Call it N_r. The probability of completing the upload of the entire page within the kth round since the start of transmitting the page is the probability that each packet in the page is successfully delivered within k rounds. Assuming independence of the losses of different packets within a page, we have

$$P(N_r \leqslant k) = \left[\sum_{j=1}^{k} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}. \quad (1)$$

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

$$E[N_r] = \sum_{i=1}^{\infty} i \cdot P(N_r = i) = \sum_{i=1}^{\infty} P(N_r \geqslant i), \quad (2)$$

$$E[N_r] = \sum_{i=1}^{\infty} (1 - P(N_r < i)) = \sum_{i=1}^{\infty} (1 - P(N_r \leqslant i-1)), \quad (3)$$

$$E[N_r] = \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right]. \quad (4)$$

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

$$E[N_{r,h}] = \min\{3 \cdot (N_p - 1) + h, N_p \cdot h\} E[N_r]. \quad (5)$$

Here $h \cdot E[N_r]$ is the number of rounds to download the first page, $3 \cdot (N_p - 1) \cdot E[N_r]$ is the number of rounds to download the rest of the pages if the network spans across more than 4 hops because of two-hop interference effect on pipelining, i.e. at any point of time, if a node at hop h receives data from hop $h-1$, no node at hop $h+1$ can send data at the same time because of collision at hop h. For networks with maximum hop separation less than 4, there is no pipelining of the code transfer and $N_p \cdot h \cdot E[N_r]$ is the number of rounds to download all the pages. Plugging Eq. (4) into Eq. (5), we get

$$E[N_{r,h}] = \min\{3 \cdot (N_p - 1) + h, N_p \cdot h\} \cdot \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right]. \quad (6)$$

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

$$T_{rep} = T_r \cdot E[N_{r,h_{max}}] = T_r \cdot \min\{3 \cdot (N_p - 1) + h_{max}, N_p \cdot h_{max}\} \cdot \sum_{i=1}^{\infty} \left[1 - \left[\sum_{j=1}^{i-1} P_s(1-P_s)^{j-1}\right]^{N_{pkt}}\right]. \quad (7)$$

We calculate the reprogramming time for (a) standalone application (one that does not perform radio communication) and (b) application that uses `GenericComm` component (provided by TinyOS) for communication. The application size is taken to be 1, 10, or 100 pages. In case (a), the increases in the size of the program image (in units of a page) are 10 and 20 respectively for Stream and Deluge, while in case (b), these increases are 1 and 11 respectively for Stream and Deluge. We use $P_s = 0.98$ and $T_r = 1$ time unit. Assuming that P_s stays constant across the three cases (Ideal, Stream, and Deluge), the reprogramming time becomes directly proportional to the number of pages (since the other factors are constant). Figs. 3 and 4 show the reprogramming time for the 10×10 grid. Clearly Stream outperforms Deluge and for applications having communication features (almost all sensor network applications have this feature), Stream is almost as fast as the ideal case.

4.2. Energy cost

Let C be the energy cost of transmitting a single packet. The energy cost of receiving packets depends on the specifics of the underlying application such as sleeping schedules. Moreover, since receiving and idle listening have almost the same

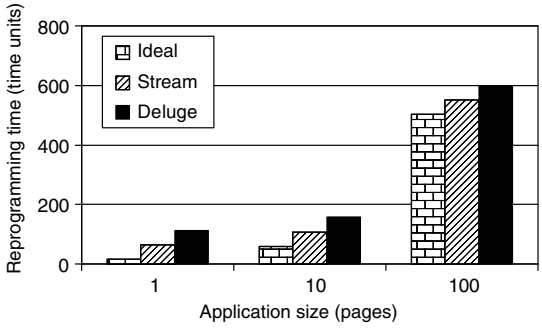

Fig. 3. Reprogramming time for 10×10 grid topology with standalone applications.

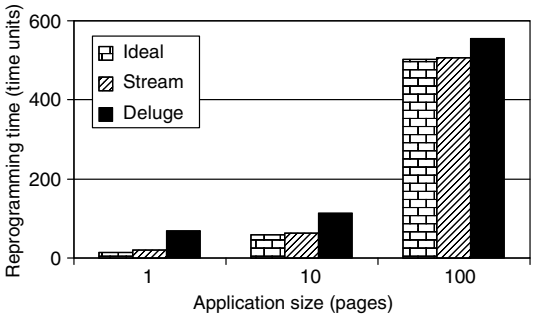

Fig. 4. Reprogramming time for 10×10 grid topology with applications having communication capability.

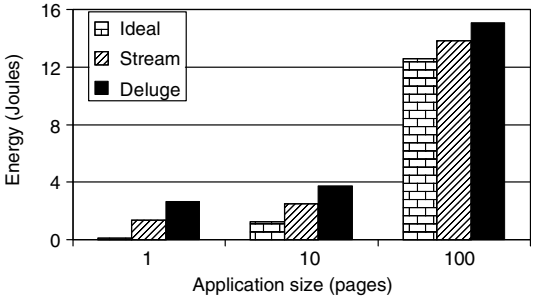

Fig. 5. Total energy consumed in the 10×10 grid topology with standalone applications.

energy cost, the energy overhead beyond packet transmission can be directly computed from the reprogramming time. So, packet transmission and writing received packets to external flash are the major sources of energy consumption during reprogramming. Both of these are functions of the number of packets transmitted during reprogramming. In this analysis, we count the number of transmitted packets and use that as an indicator of the energy performance of the two modes. This simplification is commonly done in the literature, e.g., [26]. Assuming that retransmissions of a packet are independent, the expected number of transmissions over a link for a successful transmission of a packet N_{ret} is

$$K = E[N_{\text{ret}}] = \sum_{k=1}^{k=\infty} [k \cdot (P_s(1-P_s)^{k-1})] = \frac{1}{P_s}. \quad (8)$$

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

$$E_h = K \cdot N_p \cdot N_{\text{pkt}} \cdot C \cdot \alpha_h = \frac{N_p \cdot N_{\text{pkt}} \cdot C \cdot \alpha_h}{P_s^{|S_h|}}. \quad (9)$$

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

$$E = \sum_{h=1}^{h=h_{\max}} E_h = \sum_{h=1}^{h=h_{\max}} \left[\frac{N_p \cdot N_{\text{pkt}} \cdot C \cdot \alpha_h}{P_S^{|S_h|}} \right]. \quad (10)$$

For a linear topology of N nodes with $R_{\text{tx}} = d$, where d the spacing between nodes, and R_{tx} is the transmission range, $\alpha_h = 1$, $|S_h| = 1$, and $h_{\max} = (N-1)$. For an $n \times m$ grid topology, ignoring edge effects, with $r = \sqrt{2}d$, $\alpha_h = \lceil n/2 \rceil$, $|S_h| = 3$, and $h_{\max} = (m-1)$. Let $N_{\text{pkt}} = A_{\text{pkt}} + 1 + E[N_r]$, where the second term is to account for the advertisement packet and the last term represents the expected number of request packets to successfully transmit the whole page (Eq. (4)).

We calculate total energy E expended for (a) standalone application (one that does not perform radio communication) and (b) application that uses `GenericComm` component (provided by TinyOS) for communication. The application size is taken to be 1, 10, or 100 pages. In case (a), the increases in the size of the program image (in units of a page) are 10 and 20 respectively for Stream and Deluge, while in case (b), these increases are 1 and 11 respectively for Stream and Deluge. We use fixed energy cost as 50 nJ/bit, $P_s = 0.98$, the variable (distance dependent) energy cost is 100 pJ/bit $\times r^2$, for a transmission distance of r, the receiving energy is equal to the fixed energy cost. As expected, Figs. 5 and 6 reaffirm that for the communicating applica-

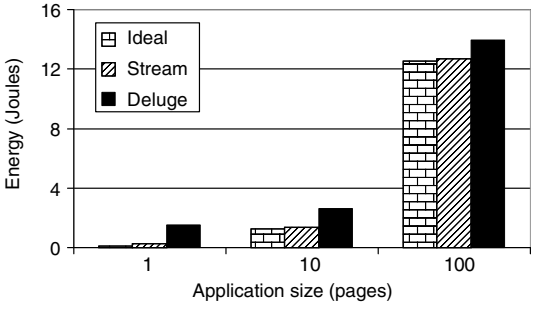

Fig. 6. Total energy consumed in the 10×10 grid topology with communicating applications.

tions, energy costs of Stream and ideal case are comparable.

5. Experiments and results

We implement Stream using the nesC programming language in TinyOS [9]. In this section, we compare the performances of Stream and Deluge for different network sizes and node densities. Both testbed experiments and simulations are used to demonstrate the advantages of Stream over Deluge. Testbed experiments are performed by using Mica2 [13] nodes and simulations are performed using TOSSIM [15], a bit level simulator for TinyOS platform. Testbed experiments show the performance of Stream and Deluge in realistic environment while simulations exhibit the scalability of these protocols.

5.1. Evaluation metrics

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

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

5.2. Testbed description and results

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

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

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

Among the various factors that contribute to the energy used in the process of reprogramming, two important ones are the amount of radio transmissions in the network and the number of flash-writes

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

As mentioned earlier, compared to Deluge the exact gain achieved by Stream in terms of number of pages transmitted depends on the user application. For a simple application which does not write to external flash and does not perform any radio communication, attaching Deluge to the user application increases the size of the application image by 20 pages whereas the increase is only 10 pages with Stream. If the user application has radio communication functionality but does not write to external flash, Stream and Deluge increase the number of pages by 1 page and 11 pages respectively. In our experiments, we use a simple application that performs radio communication but does not write to external flash. The application image alone is 11 pages, application image plus Stream-AS is 12 pages and application image plus Deluge is 22 pages.

Fig. 7 compares the average time taken by Stream and Deluge to reprogram 2×2, 3×3 and 4×4 grid networks. Interestingly, we observe from the experiments that the number of hops between two nodes is dependent on environmental conditions and changes during multiple runs of the experiment. For example, a node is sometimes able to communicate with a node separated by more than one grid point. Expectedly, the experiments show that Stream reduces the reprogramming time significantly. This large gain in reprogramming time is because Stream needs to transfer only 12 pages whereas Deluge has to transfer 22 pages. The reduction in reprogramming time becomes more pronounced for larger networks. Fig. 8 shows the total number of bytes transmitted in the network during the reprogramming period. Both data packets and control packets (request and advertisement packets for Deluge and request, advertisement and ACK packets for Stream) are considered while calculating the number of bytes. These results indicate that the energy required to reprogram the network can be greatly reduced by using Stream instead of Deluge. Although the percentage gain in reprogramming time and total number of bytes transmitted in the network vary for different topologies, in our experiments we see that Deluge requires 63–98% more reprogramming time and transfers 75–132% more number of bytes than Stream for these grid topologies (see Figs. 7 and 8).

Next we performed the experiments for linear topologies with 10 ft. distance between adjacent nodes. Source node is situated at one end of the line. Figs. 9 and 10 provide the comparison of the reprogramming time and total number of bytes transmitted in the network respectively between Stream and Deluge for different sizes of the linear topologies. Again Stream is more efficient than Deluge with respect to reprogramming time and energy used for reprogramming. In our experiments, we noticed that compared to Stream, Deluge takes 58–90%

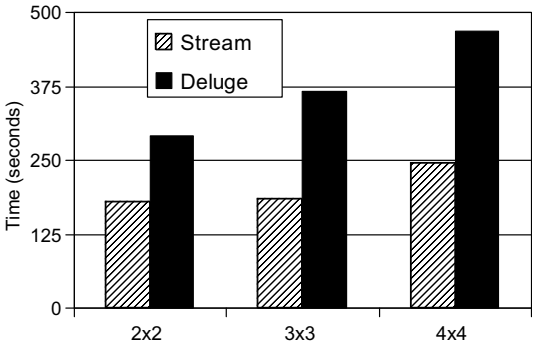

Fig. 7. Reprogramming time for grid networks (left bar is Stream, right bar is Deluge).

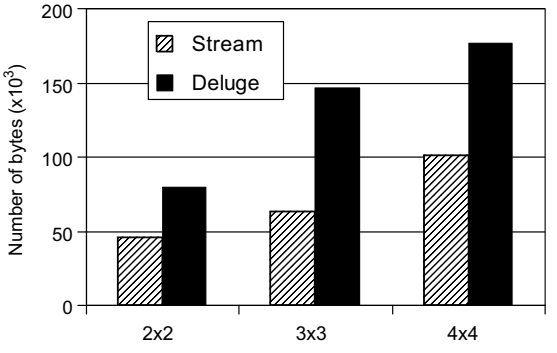

Fig. 8. Number of bytes transmitted in the network during reprogramming for grid networks.

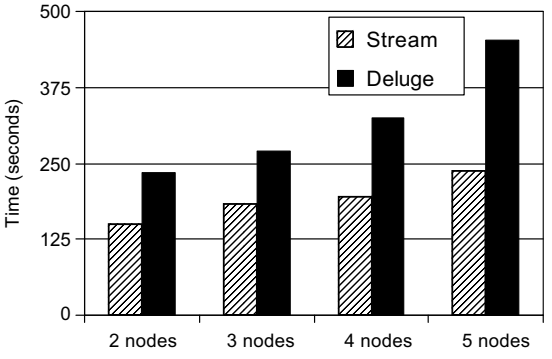

Fig. 9. Reprogramming time for linear networks.

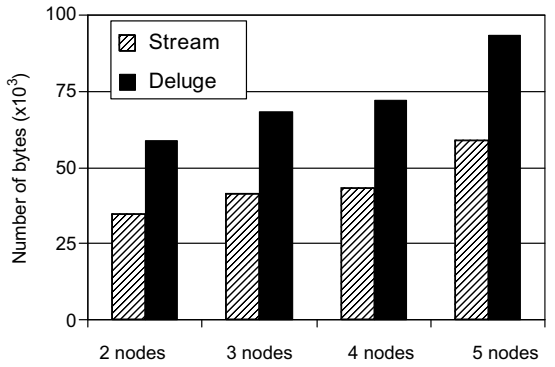

Fig. 10. Number of bytes transmitted in the network during reprogramming for linear networks.

more reprogramming time and transfers 59–70% more number of bytes for different linear topologies. If we compare the reprogramming time of 2×2 grid and linear network with 4 nodes, we find that the latter takes longer time to reprogram itself because 2×2 network can involve at most 2 hop communications (mostly 1 hop) while 4 linear nodes can have at most 3 hop communications.

The above graphs show only the number of bytes that are transmitted during the reprogramming period. In Deluge, each node keeps on broadcasting the advertisement packets even after the reprogramming period is over. As a result, the nodes have to spend energy in advertising even when reprogramming is not being done. Stream does not have this problem because as soon as the reprogramming period is over, the nodes reboot from the application image plus Stream-AS which does not broadcast advertisements. As a result, we observe a monotonically increasing difference in the number of bytes as the protocols are allowed to continue to run in the steady state.

5.3. Simulation results

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

5.3.1. Effect of network size

We use several square grid networks (10 ft. distance between successive nodes in any row and column) of varying size (up to 16×16 grid) for our simulations. A source node at one corner of the grid disseminates the user application to all other nodes in the network. Like before, Stream and Deluge need to transfer 12 and 22 pages respectively to all nodes in the network. Figs. 11 and 12 compare the reprogramming times and number of bytes transmitted in the network between Stream and Deluge for different grid sizes. It shows that both Stream and Deluge are scalable, at least up to 256 nodes simulated. In our experiments, we found that compared to Stream, Deluge requires 41–101% more reprogramming time for different network sizes.

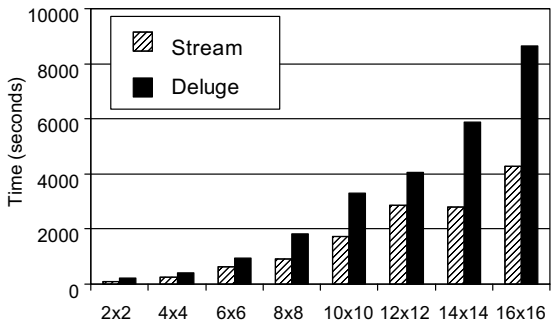

Fig. 11. Reprogramming time for $n \times n$ grids.

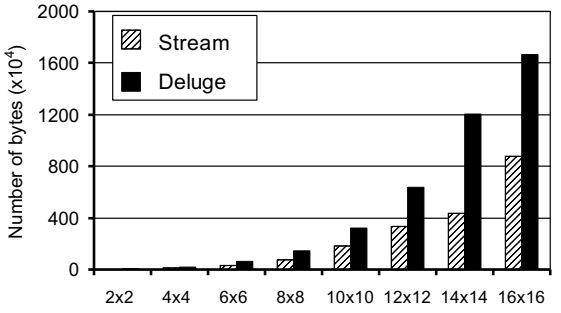

Fig. 12. Number of bytes transmitted in the network during reprogramming for $n \times n$ grids.

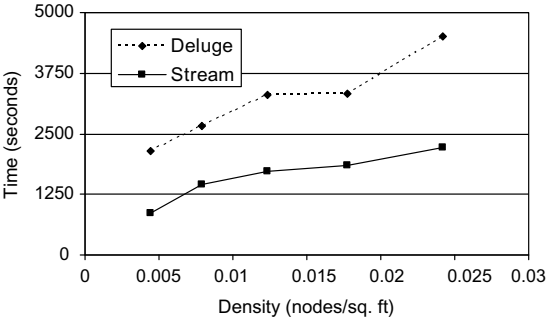

Fig. 13. Reprogramming time for different node densities.

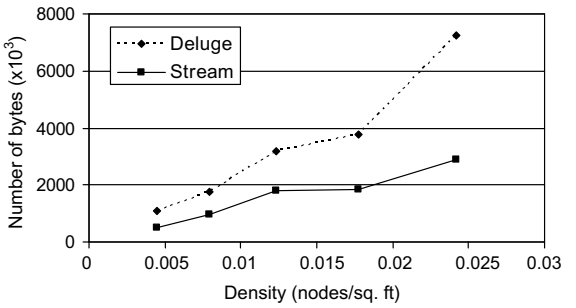

Fig. 14. Number of bytes transmitted in the network during reprogramming for different node densities.

We noticed that the increase in the total number of bytes transmitted in the network for Deluge compared to Stream was between 75% and 112% for different network sizes.

The goals of the mathematical analysis in Section 4 and simulation in this section are not to find the exact values of the reprogramming time and energy. The exact mathematical analysis is extremely difficult to do and as far as we know, there has been no such attempt in any published literature. Also TOSSIM does not simulate the node hardware and execution time exactly. So our objective in both mathematical analysis and simulation is to compare Stream and Deluge in terms of reprogramming time and energy and show the trend of these quantities as a function of the parameters like network sizes, size of the user application, etc. For example, if we compare the ratios of the reprogramming time for Deluge to that for Stream for 10×10 grid networks, it is about 1.8 from mathematical analysis and about 1.9 from simulation results. The results from simulation and analysis can be used only to observe the trend and to compare the two protocols. It would be incorrect to try to obtain accurate absolute values of any of the output parameters from either simulation results or analytical results. The absolute results are given by the testbed experiments.

5.3.2. Effect of network densities

To compare the performances of Stream and Deluge for different node densities, we vary the number of nodes in a 90 ft. $\times$ 90 ft. area. For each node density, the nodes are still arranged in grid fashion with uniform spacing between the adjacent nodes (just the spacing decreases with increasing density). Fig. 13 shows that Stream reprograms the network much faster than Deluge for all network densities and Fig. 14 shows that Stream uses lesser number of bytes than Deluge. The increase in node density increases the reprogramming time due to two reasons. First, there is an increase in the number of nodes in a given area resulting in more collisions of the transmitted packets. Second, there are simply more nodes that need to download the new application. These figures show that for higher node densities, the gap between reprogramming times as well as number of bytes between Stream and Deluge widens further. This can be explained by the fact that Stream reduces collisions more effectively due to the reduced number of bytes transferred.

5.3.3. Profile of code dissemination

Fig. 15 shows the profile of code dissemination with Stream in a 9×9 grid with 10 ft. separation. The fill-pattern of the node indicates its time to download the application code. The results indicate that the dissemination takes place uniformly with hop distance from the source (which is at the top left corner). The results are close to what we get for Deluge and matches with what the authors find in [6] for a low density network.

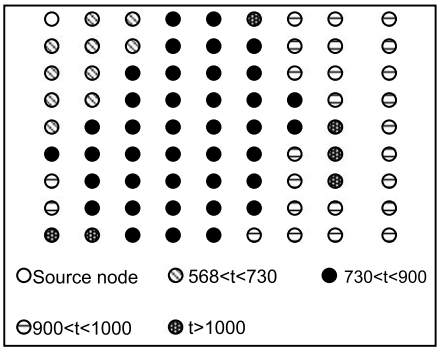

Fig. 15. Code dissemination profile according to the convergence time of a node.

6. Stream with opportunistic node sleeping

6.1. Background and rationale

Since sensor networks consist of energy-constrained nodes, one of the biggest goals of any sensor network protocol is to minimize the energy consumed by the nodes. As idle listening is a major source of energy consumption, many practical sensor network applications put the nodes to sleep as much as possible when the nodes are not doing any work, primarily, sensing and communicating. Thus the sensor network reprogramming protocol should also aim at putting the nodes to sleep mode whenever they are not actually involved in reprogramming. When the new code is injected into the network, it takes some time for the new code to reach the nodes which are far from the point of injection in the network. Freshet [8] cleverly estimates how long it takes for the new code to reach one of the neighbors of a given node and puts that node to sleep for the estimated duration. It does this by having each node estimate the number of hops it is away from the point of injection and then using an empirically calculated formula relating the time to the number of hops. Secondly, Freshet also puts the node to sleep for part of the time in the quiescent phase, i.e., when the code upload is complete. Note that Stream possesses the second advantage because immediately after the code upload is over, all nodes in the network reboot from image 1 (User application + Stream-AS) and the user application can put the node to sleep as it deems necessary. If Stream is operated in the alternate mode (to be described in this section), it can also achieve the other advantage of Freshet—putting nodes to sleep until the newly injected code arrives at one of its neighbors.

In this new mode of operation called Opportunistic Sleeping, Stream does not reboot a node from image 0 (Stream-RS) until the new code arrives at one of the neighbors of the node. A sensor node running image 1 (User application + Stream-AS) reboots from image 0 for reprogramming only if it hears an advertisement which is different from its meta-data. We will describe this mode in Section 6.2. It should be noted that in Freshet, each node *estimates* how long it takes for the code to reach one of its neighbors based on its distance (hop count) to the base node. Because of the variability of the code propagation characteristics, such an estimate may not always be accurate resulting either in decrease of energy savings (underestimate of the time for the code to reach the node) or delay in reprogramming (overestimate). However, Opportunistic Sleeping mode of Stream does not have this problem because it does not put the node to sleep for some *estimated* interval, but it causes the node to reboot from image 0 (Stream-RS) only when the new code actually arrives at one of its neighbors.

Some sensor network applications may require all the nodes in the network to be running the same version of the application at any given time. The opportunistic sleep mode of Stream's operation violates this requirement. As a result, this mode should be used only if such restrictions are not present.

6.2. Protocol description

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

1. Let α_1 be the set of nodes running image 1 which are within the communication range of the base node. Each of these nodes listens to the advertisement from the base node. The advertisement as usual carries an image version number, total number of pages in the code image and number of complete pages. If the advertisement is different than the code image version it currently has, each node in α_1 reboots from image 0. Otherwise, it does not reboot from image 0. Since the base node has newly injected image 1, all nodes

in α_1 reboot from image 1. Note that unlike in the primary mode of Stream, no reboot message needs to be broadcast through the network.
2. After nodes in α_1 reboot from image 0, they start getting pages of the new code image from the base node. The three-way handshake to get the code image pages remains unchanged from the primary mode of Stream.
3. The nodes in α_1 advertise their meta-data once they have a completed page of the code image. In line with existing multi-hop reprogramming protocols, they do not wait for the entire code image to be downloaded before advertising their meta-data. This is identical to the primary mode of Stream.
4. Let α_2 be the set of nodes which are within communication range of at least one node in α_1. When nodes in α_2 hear the new advertisement from nodes in α_1, they also reboot from image 0. In this way, the nodes reboot from image 0 for the purpose of reprogramming only when the new code image has arrived at one of its neighbors.
5. Using the same method as explained in Section 3.2, all nodes in the network get reprogrammed and reboot from image 1 (the new code image that was just downloaded).

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

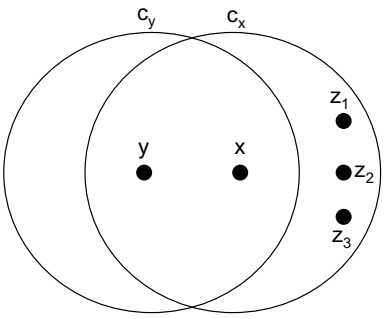

Fig. 16. Illustration of which nodes reboot for reprogramming. C_x and C_y are the communication ranges of nodes x and y respectively.

In opportunistic sleeping mode, no reboot message is explicitly sent. Instead, a node x running Stream-AS decides whether to reboot from Stream-RS based on the advertisement message it receives from say node y. If the advertisement from the node y has y.version $\neq x$.version, then the node x reboots from Stream-RS. If y.version $< x$.version, then the node x reboots with the intention of bringing node y up-to-date. However the neighbors of node x (z_1, z_2, z_3 in Fig. 16) will not reboot from Stream-RS if they are not neighbors of node y because they only hear the advertisement from node x and x.version $= z_1$.version $= z_2$.version $= z_3$.version. On the other hand, if y.version $> x$.version, node x will reboot from Stream-RS and once it gets the first page of the new image, it will start advertising the new image. Then the neighbors of node x (z_1, z_2, z_3 in Fig. 16) will also reboot from Stream-RS because they find that node x has new version of the image. As a result, those nodes also start getting reprogrammed with the new image.

6.3. Experiments and results

We performed testbed experiments and TOSSIM simulations with opportunistic sleeping mode of Stream to show how long a node can sleep before rebooting from image 0 during the initial stage of reprogramming before the newly injected code arrives at one of its neighbors. Note that the delay between the injection of the new code image and the instant when this code arrives at one of its neighbors of a given node depends on how far (hop count) the node is from the base node. As this hop distance increases, the delay also increases. The advantage of this mode of Stream is pronounced only for larger-hop networks. To do this

with the limited number of sensor nodes that we have, we ran the experiments on various linear topologies having up to 11 nodes. As shown in Fig. 17 an originator node (node 0) situated at one end of the line disseminates code to all the nodes in the network. Let the nodes be arranged as shown in Fig. 17 where the node next to node 0 is node 1, the node next to node 1 is node 2 and so on. To achieve maximum possible hops between the base node and the farthest node from the base node, we restrict the communication of a node i with node $(i-1)$ and node $(i+1)$ only. Fig. 18 shows the delay (time interval between the instant when new code is injected to the base node to the instant when the node reboots from image 0) for each node in the network as a function of its hop count from the base node. As expected, the delay increases with the hop count. Note that significant amount of energy can be saved if the nodes sleep for these intervals since the current draw in the sleep mode is three orders of magnitude less than in the idle mode for the mote class of sensor nodes (μA versus mA).

As explained above, the advantage of the new mode of Stream becomes more pronounced for larger-hop networks. So, we conducted TOSSIM simulations for linear networks of sizes up to 150 nodes. As with testbed experiments, we restrict the communication of a node i with node $(i-1)$ and node $(i+1)$ only to achieve maximum possible number of hops in the network. Fig. 19 shows the simulation results. The amount of energy saved by the opportunistic sleeping mode of Stream is quite significant for larger networks. Note that TOSSIM simulations do not provide the exact numbers and should be used only to observe the trend.

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

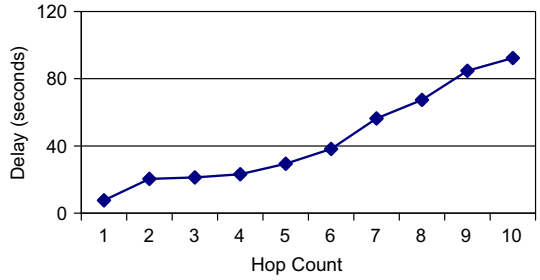

Fig. 18. Testbed result: delay before a node reboots from Stream-RS for reprogramming as a function of its hop count from the base node.

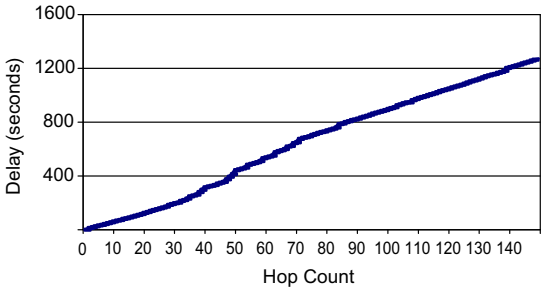

Fig. 19. Simulation result: delay before a node reboots from Stream-RS for reprogramming as a function of its distance (hop count) from the base node.

interval if the user application does not require it to be awake. In other words, depending upon the user application's sleep schedule, the nodes may not sleep for the entire delay interval shown in Figs. 18 and 19. Since most of the sensor applications sleep for a longer interval before waking up for a short interval to do its job (sensing, sending data, etc.), the sleep time will likely be close to the delay values shown in Figs. 18 and 19. The energy savings per reprogramming due to this delay is shown in Fig. 20 for different duty cycles (ratio of sleep time to one cycle consisting of sleep time and awake time) of the sensor node. Energy savings is calculated using the formula: Savings = Voltage × (Current for idle radio + Current for idle CPU) × Sleeping time (as calculated from the TOSSIM experiments) × (1-duty cycle). For mica2 platform, current for idle radio = 7.03 mA, current for idle CPU = 3.2 mA. We neglect the current draw in the sleep mode, which is less than 1 μA.

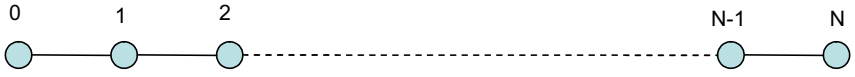

Fig. 17. Linear topology with nodes being reprogrammed using alternate mode of Stream with node 0 as the base node ($N = 1, 2, \ldots, 11$).

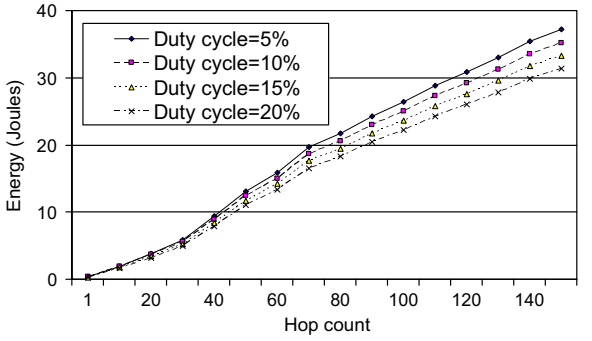

Fig. 20. Energy saving achieved by Stream over Deluge due to nodes sleeping till the code image arrives at its vicinity.

6.4. Mathematical analysis

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

$$T_{\text{round}} = T_{\text{adv}} + T_{\text{req}} + T_{\text{data}},$$

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

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

$$P(N_{\text{tx}} = k) = (1 - P_s)^{k-1} P_s.$$

The expected number of transmissions for a given packet is

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

T_{adv} can be approximated as follows:

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

where t_l is the approximate time interval between two advertisements. Note that Stream uses Deluge's advertisement policy. It divides time into intervals $[t_l, t_h]$ and each node decides whether to advertise or not in a given interval based on the number of similar advertisements it has heard in the previous interval. We take the lower value t_l because once the originator gets the new version of the code, it sets its advertisement period to t_l and the nodes hearing the advertisement from the originator also set their advertisement periods to t_l. We also assume that there were not enough similar advertisements in the previous interval to prevent the node from advertising in the current interval. GX^2 is the MAC delay for a single packet, where X is the number of contending nodes. The MAC delay is difficult to compute analytically and to the best of our knowledge, no closed form solution has yet been proposed. The authors in [16] show that for the region of interest (low contention) the delay is approximately proportional to the square of the number of contending nodes. Here G is the proportionality constant and X is the number of contending nodes. T_x is the transmission time for a single packet. T_{proc} is the processing time required by a node after receiving the packet.

T_{req} can be calculated as follows:

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

where $E[N_{\text{reqs}}]$ is the expected number of requests a node makes to complete a given page and $E[t_r]$ is the expected time between two requests. T_{data} can be calculated approximately as follows:

$$T_{\text{data}} = E[N_{\text{tx}}]N(GX^2 + T_x + T_{\text{proc}}),$$

where N is the number of packets in a page.

Fig. 21 shows the delay as a function of hop count for a grid network with $\delta = 10$ ft. separation between adjacent nodes. Probability of successful transmission of a packet P_s is taken as 0.9. For Stream, $t_l = 2$ s, $t_r = 0.5$ s and $N = 48$ packets. From [6], we take $E[N_{\text{reqs}}] = 5.4$. For mica2 node, transmission rate is 19.2 kbps and hence $T_x = 0.015$ s. We take $T_{\text{proc}} = 0.001$ s. To calculate MAC delay GX^2, we take $G = 1$ [3]. For a given node, the number of contending nodes varies with the location of the node in the network. For example, for the grid network, the nodes along the diagonal of the grid have higher number of contending nodes while those at the periphery have less contending nodes. We assume that the network is large and hence the average number of contending nodes

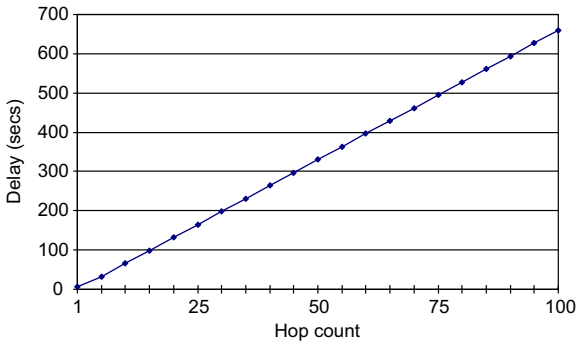

Fig. 21. Delay before a node reboots from Stream-RS for reprogramming as a function of its distance (hop count) from the base node.

is $9/4\delta^2$ (eliminating boundary effects) and the number of contending nodes is $9/4\delta^2 \times \pi r^2$ where r is the transmission range. The interference range of a node may be different from its transmission range. The difference can be easily accommodated in our analysis by replacing the communication range with the given interference range. Fig. 21 shows that the time delay before the node reboots from Stream-RS for reprogramming is quite large for the nodes distant from the originator of the code. Under the new mode of operation of Stream, the nodes can sleep for this duration, and thus Stream can conserve the energy for network reprogramming which increases with the network size. As mentioned before, results from TOSSIM simulations and mathematical analysis are not exact and should be used just to observe the trends. The trends of the delay values with increasing hop count obtained from analysis (Fig. 21), real testbed experiments (Fig. 18) and simulations (Fig. 19) are comparable.

7. Effect of user application's sleep/awake scheme on reprogramming

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

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

8. Conclusion

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

Further we are experimenting with making Stream work with multiple source nodes, ability to avoid congestion collapse in the network during high reprogramming activity, and handling heterogeneous nodes.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ECS-0330016 and the Indiana 21st Century Research and Technology Fund Grant No. 512040817. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

[1] J. Luo, P.T. Eugster, J.P. Hubaux, Route driven gossip: probabilistic reliable multicast in ad hoc networks, in: The Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), 2003, pp. 2229–2239.

[2] J. Kulik, W. Heinzelman, H. Balakrishnan, Negotiation-based protocols for disseminating information in wireless sensor networks, Wireless Networks 8 (2/3) (2002) 169–185.

[3] G. Khanna, S. Bagchi, Y.-S. Wu, Fault tolerant energy aware data dissemination protocol in sensor networks, in: The International Conference on Dependable Systems and Networks, 2004, pp. 795–804.

[4] Crossbow Tech Inc., Mote In-Network Programming User Reference, http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf, 2003.

[5] T. Stathopoulos, J. Heidemann, D. Estrin, A remote code update mechanism for wireless sensor networks, Technical Report CENS Technical Report 30, 2003.

[6] J.W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: The Proceedings of the Second International Conference on Embedded Networked Sensor Systems, Baltimore, MD, USA, 2004, pp. 81–94.

[7] S.S. Kulkarni, L. Wang, MNP: multihop network reprogramming service for sensor networks, in: The 25th IEEE International Conference on Distributed Computing Systems, 2005, pp. 7–16.

[8] M.D. Krasniewski, R.K. Panta, S. Bagchi, C.-L. Yang, W.J. Chappell, Energy-efficient, On-demand Reprogramming of Large-scale Sensor Networks, ACM Trans. Sensor Networks (TOSN). June 2007, accepted.

[9] University of California, Berkeley, "TinyOS", at http://www.tinyos.net/.

[10] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, D. Ganesan, Building efficient wireless sensor networks with low-level naming, in: The Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, Banff, Alberta, Canada, 2001, pp. 146–159.

[11] P. Levis, D. Culler, Maté: a tiny virtual machine for sensor networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002, pp. 85–95.

[12] S.R. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, TinyDB: an acquisitional query processing system for sensor networks, ACM Trans. Database Syst. 30 (1) (2005) 122–173.

[13] Crossbow Technology, Inc., MPR/ MIB user's Manual at http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf.

[14] P. Levis, N. Patel, S. Shenker, D. Culler, Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor network, in: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), 2004.

[15] P. Levis, N. Lee, M. Welsh, D. Culler, TOSSIM: accurate and scalable simulation of entire TinyOS applications, in: First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), 2003.

[16] J.H. Kim, J.K. Lee, Performance analysis of MAC protocols for wireless LAN in Rayleigh and shadow fading channels, in: IEEE Global Telecommunications Conference (GLOBECOM), vol. 1, 1997, pp. 404–408.

[17] R.K. Panta, I. Khalil, S. Bagchi, Stream: low overhead wireless reprogramming for sensor networks, in: Proceedings

of the 26th IEEE International Conference on Computer Communications (INFOCOM), May 2007, pp. 928–936.
[18] A. Tridgell, Efficient algorithms for Sorting and Synchronization, PhD Thesis, Australian National University, 1999.
[19] N. Reijers, K. Langendoen, Efficient code distribution in wireless sensor networks, in: Proceedings of the Second ACM International Conference on Wireless Sensor Networks and Applications (WSNA), 2003, pp. 60–67.
[20] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON), 2004, pp. 25–33.
[21] J. Koshy, R. Pandey, Remote incremental linking for energy-efficient reprogramming of sensor networks, in: Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN), 2005, pp. 354–365.
[22] J. Polley, D. Blazakis, J. McGee, D. Rusk, J.S. Baras, ATEMU: a fine-grained sensor network simulator, in: Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004.
[23] A. Dunkels, B. Gronvall, T. Voigt, Contiki – a lightweight and flexible operating system for tiny networked sensors, in: Proceedings of the First IEEE Workshop on Embedded Network Sensors, 2004.
[24] P.J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, K. Rothermel, FlexCup: a flexible and efficient code update mechanism for sensor networks, in: European Workshop on Wireless Sensor Networks (EWSN), 2006, pp. 212–227.
[25] A. Dunkels, N. Finne, J. Eriksson, T. Voigt, Run-time dynamic linking for reprogramming wireless sensor networks, in: Proceedings of the International Conference on Embedded Networked Sensor Systems (Sensys), 2006, pp. 15–28.
[26] S. Bandyopadhyay, E.J. Coyle, An energy efficient hierarchical clustering algorithm for wireless sensor networks, in: The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom), 2003, pp. 1713–1723.

Rajesh Krishna Panta received B.E. in Electronics from Tribhuwan University, Nepal and M.Sc. in Information and Computer Engineering from Niigata University, Japan. He is currently a Ph.D. student at Dependable and Computing Systems Lab at Purdue University. His areas of interests are wireless Ad hoc and sensor networks, applications of graph theory and stochastic geometry to network design and management.

Saurabh Bagchi joined the Department of Electrical and Computer Engineering at Purdue University in West Lafayette, Indiana as an Assistant Professor in August 2002. Before that, he did his Ph.D. from the Computer Science Department of the University of Illinois at Urbana-Champaign with Prof. Ravishankar Iyer at the Coordinated Science Laboratory. His Ph.D. dissertation was on error detection protocols in distributed systems and was implemented in a fault-tolerant middleware system called Chameleon.

Issa Khalil received the B.Sc. degree in Computer Engineering from Jordan University of Science and Technology (JUST), Jordan, in 1994, and the MS degree in Computer Engineering from JUST in 1996. He is currently pursuing a Ph.D. in the Dependable Computing Systems Lab of Prof. Bagchi S. His research interest includes key-management, secure routing protocols, and intrusion detection in Ad Hoc and Sensor networks. He has worked as the director of computer and communication center of Alquds Open University, West Bank, for more than 6 years.