Stream: Low Overhead Wireless Reprogramming for Sensor Networks

Rajesh Krishna Panta, Issa Khalil, Saurabh Bagchi
Dependable Computing Systems Lab, School of Electrical and Computer Engineering, Purdue University
465 Northwestern Avenue,West Lafayette,IN 47907
Email: {rpanta,ikhalil,sbagchi}@purdue.edu

***Abstract*— Wireless reprogramming of a sensor network is useful for uploading new code or for changing the functionality of existing code. Through the process, a node should remain receptive to future code updates because reprogramming may be done multiple times during the node's lifetime. Existing reprogramming protocols, such as Deluge, achieve this by bundling the reprogramming protocol and the application as one program image, thereby increasing the overall size of the image which is transferred through the network. This increases both time and energy required for network reprogramming. We present a protocol called *Stream* that mitigates the problem by significantly reducing the size of the program image. Using the facility of having multiple code images on a node and switching between them, Stream pre-installs the reprogramming protocol as one image and the application program equipped with the ability to listen to new code updates as the second image. For a sample application, Stream reduces the size of the program image by 10 pages (48 packets/page) compared to Deluge. Stream is implemented on the Mica2 sensor nodes and we conduct testbed and simulation experiments to show the reduction in energy and reprogramming time of Stream compared to Deluge.**

***Index Terms*— Network reprogramming, sensor networks, Deluge, three way handshake, mica2 motes.**

I. 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. A primary requirement is that the reprogramming be done while the nodes are embedded in their sensing environment. This has spurred interest in remote multihop reprogramming protocols over the wireless link. For such reprogramming, it is essential that the code update be 100% reliable and reach all the nodes that it is destined for. It is important to minimize the resource cost of the reprogramming – energy spent in disseminating the code through the network and size of the memory used on each node. The code upload should also be fast since the network's functionality is likely degraded, if not reduced to zero, during the reprogramming period.

While the cost of transmitting code is high, the cost of periodically transmitting meta-data about the code, to determine if an updated code is available, can also be high. 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 thereby necessitating retasking. The code dissemination therefore cannot be considered a one shot process and thus, it becomes important to minimize the resource consumption used in network reprogramming. Importantly, the resource cost which is incurred during the quiescent or steady state of the network[1], due to keeping the code up-to-date 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 [1], 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. Each node disseminates code to its immediate neighbor through a three-way handshake of advertisement, request, and actual code transfer. MNP and Freshet build on Deluge and optimize the transfer for energy consumption respectively 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 stable environment, the increase would be problematic. In a sensor network environment, this poses an even bigger problem. 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 –

[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 components for protocols, such as Deluge, MNP, or Freshet, and *program image* to the combined image that gets transferred over the wireless medium.

communication bandwidth, battery energy, and program Flash memory.

Our approach is optimizing *what* needs to be transferred over the wireless medium and gives rise to our protocol called *Stream*. 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 application image with minimal addition (typically 1 page). It pre-installs in each node, before deployment, the reprogramming protocol image. Stream utilizes the ability to segment the 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 just two lines of code. Stream-RS builds on Deluge and uses its three-way handshake for hop-by-hop code dissemination. Overall, Stream's design principle is to limit the size of Stream-AS and providing it the facility to switch to Stream-RS when triggered by a code update related message. The advantage afforded by Stream is demonstrated over Deluge, though it can apply to any of the three protocols, since the problem of code bloat is shared by each.

A large part of the sensor node's lifetime is spent in the quiescent state or steady-state, when it is not actively disseminating code. Hence, the energy expenditure due to exchanging control information during the steady-state is of significance. 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.

There are several challenges to implementing the basic idea of Stream in the Mica mote platform, the sensor node platform of choice today. 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, the mote platform does not provide a linking facility on the node itself. Interestingly, these constraints are also found in other common sensor node platforms, such as Sensoria's WINS and JPL's sensor node. 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.

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. We demonstrate these 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 Stream achieves 63% to 98% reduction in reprogramming time and 75% to 132% reduction in the number of bytes transferred for the grid topologies. To evaluate Stream for larger 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 II surveys related work. Section III provides the detailed design. Section IV presents the mathematical analysis. Section V explains the testbed and the simulation setups and results. Section VI concludes the paper.

II. 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]. It 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, it 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 s_a 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] aggressively optimizes the energy consumption for reprogramming. During the initial phase in Freshet, 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. Each node can go to sleep till that time thereby saving energy. Freshet also optimizes the energy consumption by exponentially reducing the meta-data rate during the quiescent phase.

III. STREAM DESIGN

A. Design Approach

Stream optimizes the number of bytes that needs to be disseminated over the wireless medium so that instead of transferring the entire reprogramming component along with the new application, only a small subset of reprogramming functionality is included in the program image. For the actual reprogramming protocol, Stream builds on the three-way handshake based code distribution seen in existing protocols. The idea is to have all nodes in the network be pre-installed with the Stream-RS (Figure 1) component that includes the complete functionality for network reprogramming. Stream-RS is installed as image 0. The application image augmented with the Stream-AS component that provides minimal support for network reprogramming is installed as image 1. Henceforth, image 0 means Stream-RS and image 1 means Stream-AS plus application image. 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 is that instead of adding the Deluge component, she adds a much smaller Stream-AS component to her application. Both are localized two-line changes to the application code.

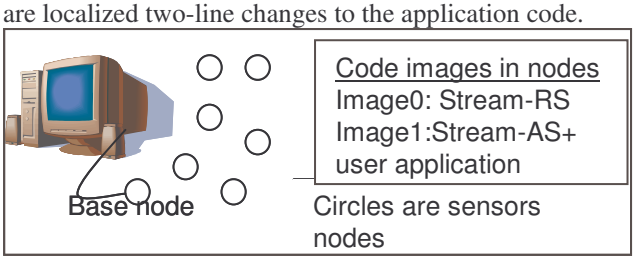

Figure 1: Images in Stream

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

B. 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 code than it currently has, it sends a request to the advertising node. Then the advertising node broadcasts the requested code pages. Each node maintains a set S containing the node ids of the nodes from which it has received the requests for code.
4. Once the node downloads the new code completely, it performs a single-hop broadcast of an ACK indicating it has completed downloading.
5. When a node α receives the ACK from a node β, it removes the id of β from its set S.
6. When the set S is empty and all the images are complete

(that is 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 are executing image 1 (Stream-AS) which has the user application.

Handling incremental node deployments. Let a node n_1 having an older version of application as image 1 and running Stream-RS join the network. Node n_1 advertises the code it has, using Stream-RS. When neighbors of node n_1 running image 1 hear the advertisement, they reboot from their image 0. Now using steps 2 through 6, the new node downloads the new application from the neighbors.

C. Design of Stream-AS

The main goal of Stream is to add small amount of reprogramming functionality to the user application instead of adding the entire reprogramming protocol (as in Deluge) so that the program image that is transferred over the wireless medium across the network is as small as possible. This is achieved by attaching the Stream-AS component to the user application. Stream-AS should be such that the increase in the size when it is attached to the user application is minimum and at the same time, the node should be receptive to code updates in the future.

Stream-AS provides the functionality to reboot from image 0 when the user gives the reboot command. This reboot command is disseminated through the network according to steps 1 and 2 in the Section Protocol Description. The 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 also 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. Once a node reboots from image 0, Stream-RS takes care of disseminating the new application image.

As mentioned above, 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 the user application to Stream-AS instead, replace `DelugeC` by `StreamASC`.

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. As a result, radio resources are continuously used by Deluge even in the steady state. However, in Stream, in the steady-state, each node is running Stream-AS, which does not proactively advertise the code image that it has. However, both new nodes joining the network and new code pushed in by the base station are handled. The nodes running user application plus Stream-AS in the steady state receive the advertisement from the new nodes, reboot from Stream-RS, and send the updated application to the new nodes. When the base station has to push an updated application image, 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. The steady-state advertisements in Deluge mean that the user application has to share the node's radio resources with Deluge while this is not the case when Stream is used. Also, the steady-state 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.

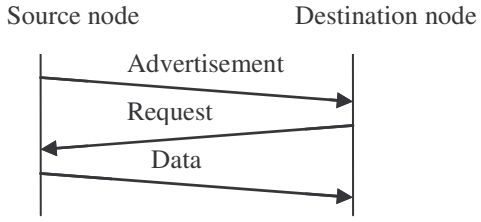

Figure 2: Three-way handshake for data dissemination

D. Design of Stream-RS

Stream-RS, preinstalled in all nodes as image 0 and executed during the 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 (Figure 2) in which each node broadcasts the advertisement about the code pages that it currently has. A node, upon hearing the advertisement of newer code than it currently has, sends a request to the advertising node. 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.

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, other nodes may still be dependant on it for getting the updated code image. Therefore the node needs to continue to run Stream-RS. To handle this, when a node receives a request for code, it puts the node-id of the requesting node in the set S. When a node completes downloading the new application image, it broadcasts an ACK. 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 for the set S of a node A:

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

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

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

The first condition is that no neighbor is waiting on A to send it updated code 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 B would never be included in A's set S. Let a node n_1 hears the advertisement of newer data than it currently has from node n_2 during the reprogramming phase. Before n_1 sends request for the new data, some other one-hop neighbor of n_2, say n_3, may send the request. In response to the request from node n_3, n_2 broadcasts the code. So, n_1 may never send the request to n_2 but keep on receiving the code from n_2, triggered by the request from n_3. If all the nodes that explicitly request data from the advertising node n_2 complete downloading the new application earlier than n_1, node n_2 will reboot from the new application. This leaves 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 because after the advertising node n_2 has rebooted from the user application, it still can hear the advertisement sent by n_1. Upon hearing the advertisement, n_2 will reboot from Stream-RS and start sending code to n_1 through the three-way handshake.

These changes in Deluge ensure that all nodes reboot from the user application after reprogramming is done. In Deluge, in contrast to the automatic operation in Stream, 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 base node.

IV. STREAM ANALYSIS

A. Energy Cost

Here we analyze the 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, each page has N_{pkt} packets and each packet has N_b bytes. Let C be the energy cost of transmitting and receiving a packet once and P_s be the probability of successful transmission of a packet over a single hop. Assuming that retransmissions of a packet are independent, the probability that the number of retransmissions (N_{ret}) of a packet equals k is given by

$$P(N_{ret} = k) = P_s(1-P_s)^{k-1} \quad (1)$$

The expected number of retransmissions $K = E[N_{ret}]$ is

$$K = E[N_{ret}] = \sum_{k=1}^{k=\infty}\left[k \cdot \left(P_s(1-P_s)^{k-1}\right)\right] = \frac{1}{P_s} \quad (2)$$

The energy cost of a node at h hop from the source to completely download the application is given by

$$E_h = K \cdot h \cdot N_p \cdot N_{pkt} \cdot C = \frac{h \cdot N_p \cdot N_{pkt} \cdot C}{P_s} \quad (3)$$

The total energy overhead of uploading the application on all the nodes in a network in which each node is at most h_{max} hops from the source is given by

$$E = \sum_{h=1}^{h=h_{\max}} N_{Nh} \cdot E_h = \sum_{h=1}^{h=h_{\max}}\left[N_{Nh} \cdot \frac{h \cdot N_p \cdot N_{pkt} \cdot C}{P_s}\right] \quad (4)$$

where N_{Nh} is the number of nodes at hop h. N_{Nh} depends on network topology and density d. For a line topology, $N_{Nh} = 1$. For a uniformly distributed network on a disk with communication radius r, $N_{Nh} = \pi dr^2(2h-1)$. For nxn grid, $N_{Nh} = h+1$ when $1 \le h \le (n-1)$ or $N_{Nh} = 2n-h-1$ when $n \le h \le (2n)$. For 10x10 grid, 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. 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. Figure 3 shows the significant difference in energy expended between Stream and Deluge. Figure 4 reaffirms that for communicating applications, energy costs of Stream and the ideal case are comparable.

B. Convergence Time

In this section, we analyze the convergence time, i.e., the time to reprogram the entire network. Let $P_k = P(N_r = k)$ be the probability that a whole page has been successfully sent in the k^{th} round, where N_r is the number of rounds. The probability that a given packet fails to be sent within k rounds is given by

$$P_{f(k)} = (1-P_s)^k \quad (5)$$

Then, the probability of successfully sending the whole page in at most k rounds ($N_r \le k$) is

$$P(N_r \le k) = \sum_{i=1}^{k} P_i = \left[1-(1-P_s)^k\right]^{N_{pkt}} \quad (6)$$

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

$$E[N_r] = \sum_{i=1}^{\infty} i \cdot P_i = \sum_{i=1}^{\infty} i \cdot P(N_r = i) = \sum_{i=1}^{\infty} P(N_r \ge i) \quad (7)$$

$$E[N_r] = \sum_{i=1}^{\infty}\left(1-P(N_r < i)\right) = \sum_{i=1}^{\infty}\left(1-P(N_r \le i-1)\right) \quad (8)$$

$$E[N_r] = \sum_{i=1}^{\infty}\left[1-\left(1-(1-P_s)^{i-1}\right)^{N_{pkt}}\right] \quad (9)$$

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

$$E[N_{r,h}] = (N_p - 1)E[N_r] + h \cdot E[N_r] = (N_p + h - 1)E[N_r] \quad (10)$$

The last term is the time it takes to download the first page, and the first term is the time it takes to download the rest of the pages. Plugging (9) into (10), we get

$$E[N_{r,h}] = (N_p + h - 1)\sum_{i=1}^{\infty}\left[1-\left(1-(1-P_s)^{i-1}\right)^{N_{pkt}}\right] \quad (11)$$

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

$$T_{conv} = T_r \cdot E[N_{r,h_{\max}}]$$

$$T_{conv} = T_r \cdot (N_p + h_{\max} - 1) \cdot \sum_{i=1}^{\infty}\left[1-\left(1-(1-P_s)^{i-1}\right)^{N_{pkt}}\right] \quad (12)$$

Assuming that P_s stays constant across the three cases (Ideal, Stream, and Deluge), the time becomes directly proportional to

the number of pages (since other factors are constant). Figure 5 and Figure 6 show the convergence times for the 10×10 grid. In reality, since Stream puts less pressure on the bandwidth than Deluge, P_s will be higher for Stream thus giving it additional advantage for convergence time.

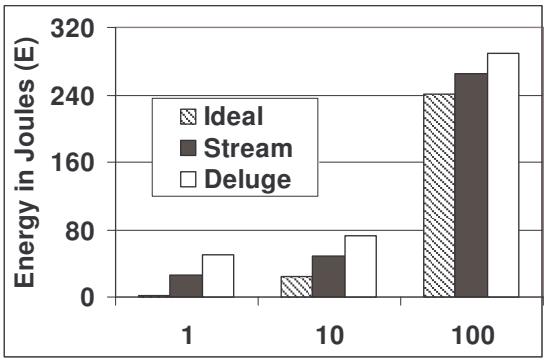

Figure 3: Total energy consumed in the 10×10 grid topology with standalone applications

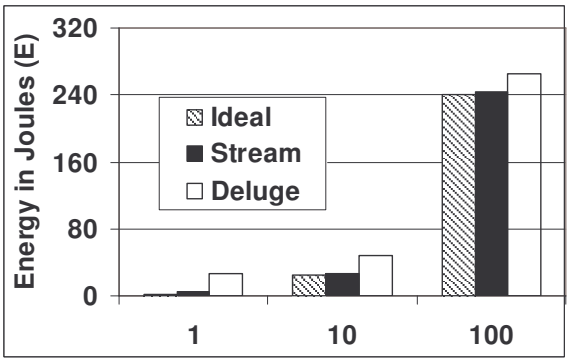

Figure 4: Total energy consumed in the 10×10 grid topology with communicating applications

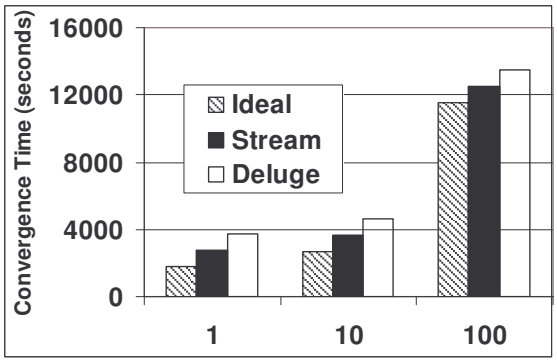

Figure 5: Convergence time for 10×10 grid topology with standalone applications

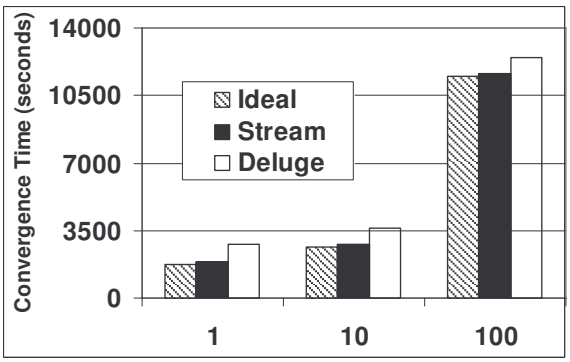

Figure 6: Convergence time for 10×10 grid topology with communicating applications

V. Experiments and results

We implement Stream using the nesC programming language in TinyOS [5]. In this section, we compare the performances of Stream and Deluge for different network sizes and node densities. Both testbed experiments using Mica2 [4] motes and simulations using TOSSIM [15] (a bit level simulator for TinyOS platform) are used to demonstrate the advantages of Stream over Deluge. Testbed experiments show the performance of Stream and Deluge in realistic environment while simulations exhibit their scalability.

A. Evaluation Metrics

Any network reprogramming protocol must ensure that *all* nodes in the network receive the application image completely minimizing the *time* and the *energy* for the reprogramming. 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.

B. Testbed description and results

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

The first set of experiments is performed in 2x2, 3x3, and 4x4 square grid networks having a distance of 10 ft between adjacent nodes in each row and column. Experiments of network reprogramming using Stream are carried out by pre-installing Stream-RS as image 0 and same version of application image plus Stream-AS as image 1 on all nodes in the network. A new application image plus Stream-AS is injected into the source node (situated at one corner of the grid) via a computer attached to it. Then the source node starts disseminating the new application image to the network. Experiments with Deluge are performed similarly by installing Deluge as image 0 and the application image plus Deluge as image 1. A new application image plus Deluge is injected into the network.

Time to reprogram the network is the time interval between the instant t_0 when the source node sends the first data packet to the instant t_1 when the last node (the one which takes the longest time to download the new application) completes downloading the new application. Since clocks maintained by the nodes in the network are not synchronized, we cannot take the difference between the time instant t_1 measured by the last node and t_0 measured by the source node. Although a synchronization protocol can be used to solve this issue, we do not use it in our experiments because we do not want to add to the load in the network (due to synchronization messages) or the node (due to the synchronization protocol). Instead, once each node completes downloading the new application image, it sends a special packet to the source node saying that it has completed downloading the new application. The source node measures the time instant t_1' when it receives such packet, timestamps the packet with t_1' and sends the packet to the computer. If the network has n nodes including the source node, the computer attached to the source node receives one t_0

and *(n-1)* number of t_1's. We take $t_{prog} = \max_{t_1'} (t_1' - t_0)$ as the reprogramming time. It should be noted that the actual reprogramming time is $\max_{t_1'} (t_1' - t_0 - t_d)$ where t_d is the time required to send the special packet from the last node to the source node. Since t_d is negligible compared to the reprogramming time, our formula is a reasonable approximation to the actual reprogramming time. Furthermore, since we are interested in the difference between the reprogramming times of Stream and Deluge, the effect of t_d cancels out.

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 summing 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. 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.

Figure 7 compares reprogramming times of Stream and Deluge for 2x2, 3x3, and 4x4 grid networks. Interestingly, we observe from the experiments that the number of hops between two nodes is dependent on environmental conditions and changes during multiple runs of the experiment. For example, a node is sometimes able to communicate with a node separated by more than one grid point. Expectedly, the experiments show that Stream reduces the reprogramming time significantly. This large gain in reprogramming time is because Stream needs to transfer only 12 pages whereas Deluge has to transfer 22 pages. The reduction in reprogramming time becomes more pronounced for larger networks. Figure 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. The results indicate that despite the additional ACK traffic in Stream, the reduction in data packets causes Stream to outperform Deluge. We see that Stream achieves 63% to 98% reduction in reprogramming time and 75% to 132% reduction in the total number of bytes transferred for these grid topologies.

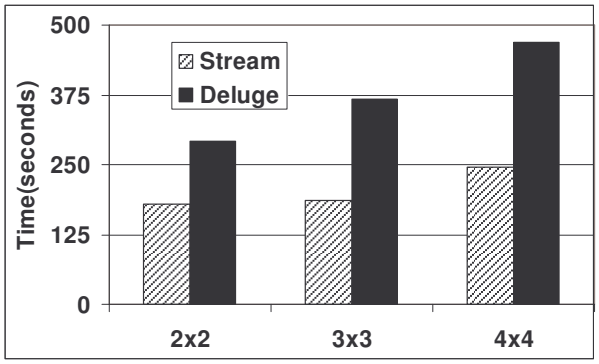

Figure 7: Reprogramming time for grid networks

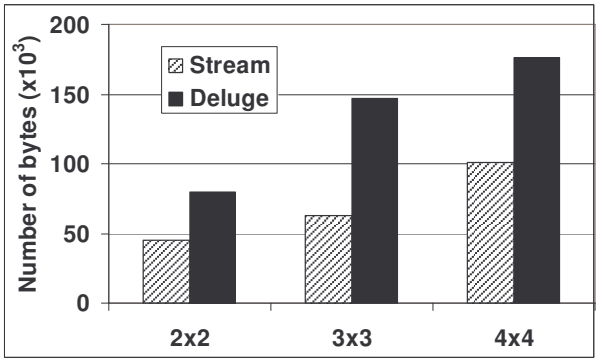

Figure 8: Number of bytes transmitted in the network during reprogramming for grid networks

Next we performed the experiments for linear topologies with 10 ft. separation between adjacent nodes. Source node is situated at one end of the line. Figure 9 and Figure 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. Stream reduces reprogramming time by 58% to 90% and the total number of bytes transferred by 59% to 70% for different linear topologies. If we compare the reprogramming time of 2x2 grid with a 4 node linear network, we find that the latter takes longer time to reprogram itself because 2x2 network can involve at most 2 hop communications (mostly 1 hop) while 4 linear nodes can have at most 3 hop communications.

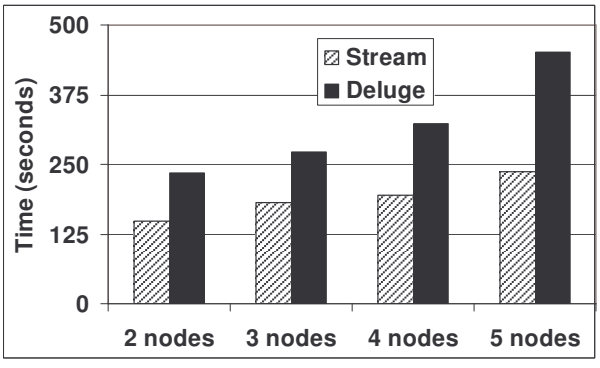

Figure 9: Reprogramming time for linear networks

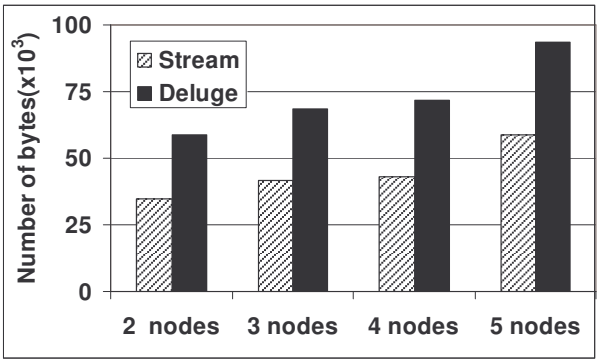

Figure 10: Number of bytes transmitted in the network during reprogramming for linear networks

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.

C. Simulation Results

In order to demonstrate the scalability of Stream and to compare it with Deluge for larger network sizes, we performed 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 is not of serious concern because we are interested in the comparison of performances of Stream and Deluge and not on the absolute values.

Effect of network size

We use several square grid networks (10 ft distance between successive nodes in any row and column) of varying size (up to 16x16 grid) for our simulations. A source node at one corner of the grid disseminates the user application to all other nodes in the network. Like before, Stream and Deluge need to transfer 12 and 22 pages respectively to all nodes in the network. Figure 11 and Figure 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 Deluge, Stream reduces the reprogramming time by 41% to 101% for different network sizes. We noticed that the reduction in the total number of bytes transmitted in the network was between 75% to 112% for different network sizes.

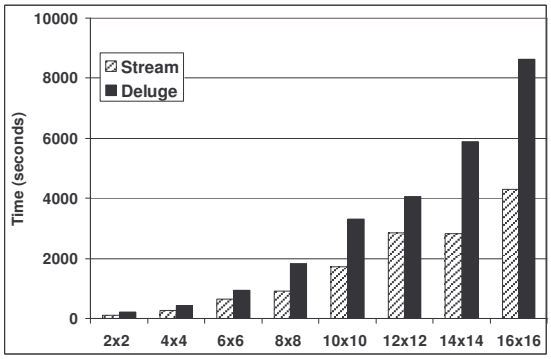

Figure 11: Reprogramming time for nxn grids

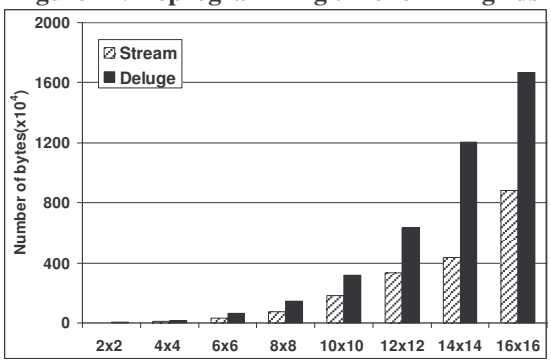

Figure 12: Number of bytes transmitted in the network during reprogramming for nxn grids

Effect of network densities

To compare the performances of Stream and Deluge for different node densities, we vary the number of nodes in a 90 ft by 90 ft area. For each node density, the nodes are still arranged in grid fashion with uniform spacing between the adjacent nodes (just the spacing decreases with increasing density).

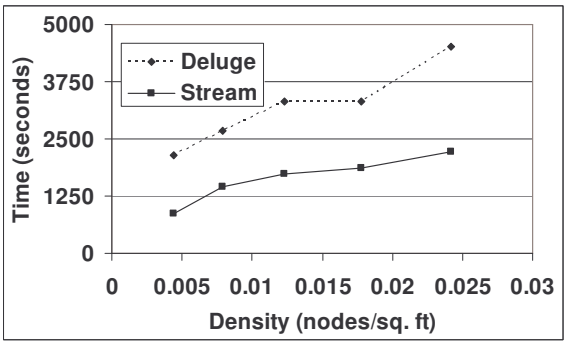

Figure 13: Reprogramming time for different node densities

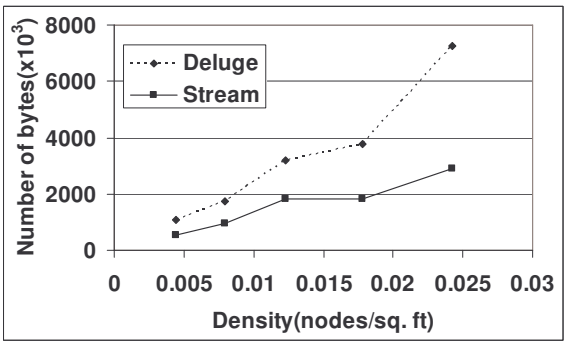

Figure 14: Number of bytes transmitted in the network during reprogramming for different node densities

Figure 13 shows that Stream reprograms the network much

faster than Deluge for all network densities and Figure 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.

Profile of code dissemination

Figure 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.

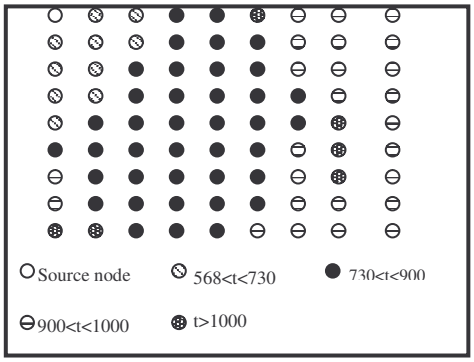

Figure 15: Code dissemination profile according to the convergence time of a node

VI. 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, the number of bytes transferred, the energy expended, and the usage of program memory. Stream is implemented on TinyOS for the Mica2 sensor node. Experiments conducted on a testbed of Mica2 motes demonstrate up to 98% reduction in reprogramming time and up to 132% reduction in the number of bytes transferred compared to Deluge. Simulation experiments in TOSSIM show the increasing advantages of Stream over Deluge with larger network sizes.

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 integration with Freshet to provide a highly energy optimized protocol.

Acknowledgment

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

References

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

[2] J. Kulik, W. Heinzelman, and H. Balakrishnan, "Negotiation-based protocols for disseminating information in wireless sensor networks," Wireless Networks, vol. 8, no. 2/3, pp. 169-185, 2002.

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

[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, and D. Estrin, "A remote code update mechanism for wireless sensor networks," Technical Report CENS Technical Report 30, no., 2003.

[6] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," at the Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA, pp. 81-94, 2004.

[7] S. S. Kulkarni and W. Limin, "MNP: Multihop Network Reprogramming Service for Sensor Networks," at the 25th IEEE International Conference on Distributed Computing Systems, pp. 7-16, 2005.

[8] M. D. Krasniewski, S. Bagchi, C-L. Yang, W. J. Chappell, "Energy-efficient, On-demand Reprogramming of Large-scale Sensor Networks," Submitted to IEEE Transactions on Mobile Computing (TMC). Available as Purdue ECE Technical Report TR-ECE-06-02, 2006.

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

[10] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, "Building efficient wireless sensor networks with low-level naming," at the Proceedings of the eighteenth ACM symposium on Operating systems principles, Banff, Alberta, Canada, pp. 146-159, 2001.

[11] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, no., pp. 85-95, 2002.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," ACM Trans. Database Syst., vol. 30, no. 1, pp. 122-173, 2005.

[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, and D. Culler, "Trickle: A Self-Regulating Algorithm for Code Propogation and maintenance in Wireless Sensor Network," Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), no., 2004.

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