Energy-efficient, On-demand Reprogramming of Large-scale Sensor Networks

MARK D. KRASNIEWSKI, RAJESH KRISHNA PANTA, SAURABH BAGCHI
DEPENDABLE COMPUTING SYSTEMS LAB
CHIN-LUNG YANG, WILLIAM J. CHAPPELL
RF SYSTEMS LAB
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING, PURDUE UNIVERSITY
EMAIL: (MKRASNIE, RPANTA, SBAGCHI, CYANG, CHAPPELL)@PURDUE.EDU

As sensor networks operate over long periods of deployment in difficult to reach places, their requirements may change or new code may need to be uploaded to them. The current state of the art protocols (Deluge and MNP) for network reprogramming perform the code dissemination in a multi-hop manner using a three way handshake whereby meta-data is exchanged prior to code exchange to suppress redundant transmissions. The code image is also pipelined through the network at the granularity of pages. In this paper we propose a protocol called Freshet for optimizing the energy for code upload and speeding up the dissemination if multiple sources of code are available. The energy optimization is achieved by equipping each node with limited non-local topology information, which it uses to determine the time when it can go to sleep since code is not being distributed in its vicinity. The protocol to handle multiple sources provides a loose coupling of nodes to a source and disseminates code in waves each originating at a source, with mechanism to handle collisions when the waves meet. The protocol's performance with respect to reliability, delay, and energy consumed, is demonstrated through analysis, simulation, and implementation on the Berkeley mote platform.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design— Network communications, Wireless communication.
General Terms: Algorithm, Design, Performance, Reliability
Keywords: Wireless communication, Sensor networks, Network reprogramming, Deluge, Three way handshake.

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 version of existing code or retasking the existing code with different sets of parameters. We use the term *code upload* for referring to both. 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 reprogramming, it is essential that the code update be 100% reliable and reaches all the nodes that it is destined for. The code upload should be fast

This material is based upon work supported by the National Science Foundation under Grant No. ECS-0330016 and 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 authors and do not necessarily reflect the views of the sponsors.
Authors' addresses: Mark D. Krasniewski, Southwest Research Institute, TX 78209; All other authors, School of Electrical and Computer Engineering, Purdue University, IN 47907. The work reported here was done by Mr. Krasniewski while he was a graduate student at Purdue University.

since the network's functionality is likely degraded, if not reduced to zero, during the period when the nodes are being reprogrammed. It is also important to minimize the resource cost of the reprogramming and querying for availability of new code. It is conceivable that code upload will be infrequent for many deployments and it may appear resource consumption is a non-issue. However, as has been noted in [1], while the cost of transmitting code is high, the cost of periodically transmitting code meta-data (e.g., for querying current version of code) also be high. Applications such as Tiny Diffusion [2], Maté [3], and TinyDB [4], use concise, high-level virtual code representations to give programs that are 20-400 bytes long. The sensor network environment has inherent unreliability in the network links due to interference, fading, as well as mobility and unreliability in the nodes which may have transient failures. Also new nodes may join the network and need code updates. The code dissemination therefore must be a continuous rather than a one shot process. Due to these reasons, resource consumption, mainly bandwidth and communication energy, becomes an important issue. There is also resource cost for a node to query for new code that may be injected into the network at any given time. This resource cost incurred during the steady state of the network must be optimized since that is the dominant phase in the network lifetime.

The underlying model for the class of network reprogramming protocols is that the binary image to be transmitted to the nodes has monotonically increasing version numbers. The image is segmented into pages (typical size 1104 Bytes) and each page is sent using multiple packets (typical size 36 Bytes). To start off, there are only a few sources of the binary image, e.g., base stations located in the sensor field. The code progressively ripples through the network with the exchange happening between neighbors through a three way handshake of advertisement, request, and actual code transfer. The advertisement and the request will collectively be referred

to as meta-data. The meta-data is typically much smaller in size than the data (the code) and is used to suppress redundant data transmission. The advertisement indicates availability of code at the sender, the request indicates that some or all of the advertised pages are needed at the sender, following which the actual code transfer takes place in units of pages which are sent as packets.

In this paper, we present a protocol called *Freshet*[1], which fits in this genre of protocols. The first realization is that a brute force flooding method is not feasible due to the enormous bandwidth overheads. In view of limited bandwidth resources and the energy consumption due to communication, it is important to suppress redundant transmissions of the data and the meta-data. The suppression uses the shared nature of the wireless medium and the capacity of a node to overhear its neighbors' communication. For example, if a node A in the network has version v and a neighbor node B requests pages of version v' ($< v$) from a node C, then A can proactively send the more recent code to B. This will cause a suppression of the transmission from C to B if C and A are neighbors. Next, we use pipelining of the different pages in a binary image to expedite the code upload. Each interested node may initiate the process of forwarding the code in units of a page as it receives the pages and aggregates them to create its own complete binary image. This is in contrast to the approach in Mote Over the Air Programming (MOAP) [5] where the forwarding happens only when the entire code has been assembled at a node. Since a binary image may consist of many pages and the wireless links are failure prone, the MOAP approach may lead to excessive retransmissions and therefore bandwidth overheads. Freshet can also speed up the process when multiple sources of code are available. The key insight to enable this

[1] OED: *Freshet* – (i) A small stream of fresh water (Obs. exc. poet.); (ii) A stream or rush of fresh water flowing into the sea; (iii) A flood or overflowing of a river caused by heavy rains or melted snow.
Used by Bowen in *Virgil* as "A cave … sweet fountain freshets within it."

is to allow nodes to receive pages out of sequence for streams from different sources. This leads to somewhat more state maintenance at the node but substantially speeds up the process.

Freshet has the design goal of reducing the energy consumption due to code upload. For this, it attacks the single biggest source of energy drain – idle listening energy. A fundamental insight used in Freshet is that nodes can be put to sleep by making the advertisement-request-data handshake happen only at certain points in time. When new code is introduced into the network, Freshet has an initial phase, the *blitzkrieg phase*, when information about the code propagates through the network rapidly along with some topology information. The topology information is used by each node to estimate 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. The potential for energy savings grows with the size of the network. Freshet also optimizes the energy consumption by exponentially reducing the meta-data rate during conditions of stability in the network (the *quiescent phase*) when no new code is being introduced.

In order to demonstrate the behavior of Freshet, we build simulation models in TOSSIM, which is a discrete event network simulator that compiles directly from unmodified TinyOS application code [22]. TOSSIM captures the behavior of the entire TinyOS network stack in a detailed manner and is used to solve the problem of scaling of our actual sensor network testbed. TOSSIM does not represent actual real-world performance of a sensor network, but its primary use in this paper is to compare the performance of Freshet and Deluge, so with both protocols running TOSSIM any simulation discrepancies should affect each protocol similarly. We also present performance results from a small sized implementation testbed illustrating the

advantages of Freshet over the state-of-the-art Deluge in terms of energy and that Freshet does not significantly increase the reprogramming time.

It must be noted that in some of the high level goals and design approach, Freshet has similarities with two recent protocols – Deluge [6] and MNP [7]. However, there are substantial differences in the protocol design which lead Freshet to make the following novel contributions.

1. Freshet shows that adding limited network topology information to local information provides energy benefits while preserving scalability.
2. Freshet addresses the problem of code upload from multiple original sources. It shows the benefit of using interleaved transmission of pages to speed up the code upload process in the multiple source situation.
3. Freshet shows a method for energy optimization in the quiescent phase while preserving the reliability guarantee of other protocols.

The rest of the paper is organized as follows. Section 2 presents a survey of related work. Section 3 presents the basic design of Freshet and Section 4 three extensions. Section 5 discusses coordinating the Freshet sleep schedule with user applications. Section 6 presents the analysis and Section 7 the experimental results. Section 8 concludes the paper.

2 Related Work

The field of network reprogramming in the large scale wired distributed systems has focused on the problem of reliability and efficient utilization of bandwidth. For example, [8] provides methods for efficiently computing increments to the update. They have not dealt with resource constraints on the nodes themselves. Due to the wired environment, the solutions do not have the ability to leverage overhearing neighbor communication.

In a large scale wireless network, data dissemination through unregulated flooding using broadcast by each node is known to cause a broadcast storm [9], thereby limiting the scalability of such a solution. Hence, researchers have proposed randomized tree based multicast protocols with the source at the root of the tree, receivers at the leaves, and intermediate nodes responsible for local recovery at the intervening levels of the tree. Scalable Reliable Multicast (SRM) [10] is an important protocol in this class. In SRM, when a member detects a message loss, it initiates a recovery procedure by multicasting a retransmission request in the local region. Any member having the desired message in its cache responds by multicasting the message, with a back off mechanism being used to prevent redundant requests and replies. The idea of suppression through deferred messages in Freshet comes from SRM. Further scalability 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 [11]. The probability of the update reaching all the group members 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, including 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 [12] and SPMS [13] 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. Freshet borrows the idea of hop-by-hop NACK based error recovery present in many protocols proposed for wireless sensor networks (WSNs), such as Garuda [14].

There are four major sensor network reprogramming approaches that have appeared in the literature. TinyOS [15] includes limited support for network programming via XNP [16]. However, XNP only operates over a single hop and does not provide incremental updates of the code image. The Multihop Over the Air Programming (MOAP) protocol extends this to operate over multiple hops [5]. MOAP introduced several concepts which are used by later protocols, including Freshet, namely, local recovery using unicast NACKs and broadcast of the code. However, MOAP does not leverage the pipelining effect with segments of the code image. The two protocols that are substantially more sophisticated than the rest are Deluge [6]and MNP [7]. Both use the three way handshake and the segmentation into pages and packets. Deluge builds on top of Trickle [1], a protocol for a node to determine when to propagate code in a one hop case. Deluge leverages overheard advertisements or requests to decide when to create a new advertisement or send a new code update. MNP is a more recent protocol whose design goal is to choose a local source of the code which can satisfy the maximum number of nodes. The authors provide a detailed algorithm for sender selection using the number of requests seen by a sender as the key parameter for the selection. They provide energy savings by turning off the radio of all the nodes that are not selected as the sender. While this protocol does provide advantages, in [23] it was shown that MNP downloaded code significantly more slowly than Deluge. Therefore, this paper will focus on comparing Freshet to Deluge's performance.

Freshet, while it shares most of the design goals and some design features of Deluge and MNP, is different in many important aspects. To elaborate and paraphrase the key differences

mentioned in Section 1, Freshet optimizes the energy consumption more aggressively through turning off the nodes between the blitzkrieg phase and the distribution phase using limited topology information. It also trades off the responsiveness of the protocol to newly joining nodes for saving further energy during the steady state. It also uses out of order paging to speed up the code update with multiple sources of the code.

More recently, our protocol called Stream [28] optimizes the energy of wireless reprogramming by limiting *what* is sent wirelessly across the network. Rather than sending the reprogramming protocol image together with the application image, it sends only the application image together with a small control protocol image, while it pre-installs the bulk of the reprogramming protocol in the node before deployment.

3 Design of Freshet

3.1 System Model

For a quick reference of the meanings of important parameters used in Sections 3 and 4, please see Table 1.

Parameter	Meaning
v	Version number of the code
p	Current page of the code
p_{max}	Maximum (total) page number of the code
w	If a node hears more than w warning messages, it does not send the warning message.
t_{off}	Time for the three way handshake between two neighbor nodes
b_n	Number of neighbors of a given node
τ	In quiescent phase, each node listens for a period of $\tau/2$ and then decides with probability $1\text{-}1/b_n$ that is should sleep for the next $\tau/2$ period
numSrc	Number of originators

Table 1 : Meanings of some important parameters

Initially, a few specialized nodes, such as base stations, have the entire code image. These nodes are called *originators*, to distinguish them from *sources* of the code, since any node can act as a source as soon as it has received a subset of the code image. The binary code image is segmented into equal sized pages and each page is split into multiple packets. The code is transferred through the links in units of a packet while the three-way handshake happens in units

of a page. Each new image injected into the network has a version number attached to it, which increases monotonically. A node obtains code through monotonically increasing page numbers. When a node hears of code for a later version, it suspends any transfers for the code of the earlier version. Each node maintains local state of tuples of (v, p, p_{max}) where v gives the version number, p the current page with the node, and p_{max} is the maximum page number. Thus looking at a code image transfer packet, a node can uniquely determine if it needs the packet.

Freshet uses spatial multiplexing to transfer the code. This implies that a node can transfer the code to a neighbor before it has received all the pages for a given version. In effect, the node can initiate transfer once it has the first page for the version. This makes the delay proportional to the sum of the network diameter and the code size rather than the product of the two.

Now we describe the three phases in Freshet that each node goes through.

3.2 Blitzkrieg Phase

In the blitzkrieg phase, Freshet propagates information about the nature of the new code to all nodes in the network. This is accomplished through a fourth type of message, a *warning message*, apart from the advertisement, request, and broadcast data messages. This message contains information about the new code in the form of the version number, the number of pages, and how far the sending node is from the data source, in terms of hop counts. The blitzkrieg phase enables energy optimization since each node can use the hop count information to determine when it will enter the distribution phase.

The pseudo-code showing the operation of the blitzkrieg phase is shown in the Appendix in Figure 27. The hop count is incremented by each intermediate node routing the warning message. Every time a node hears a unique warning message with code information more recent than its own, it starts a short, randomized timer. Once this timer fires, and the node has not heard

more than *w* warning messages with the same code version as its own, it sends out the warning message. The node sends the exact same message as the one it first received, except that it increments the hop count from the original message. This information therefore gives the receiver an estimate of how many intervening nodes from the node have the data and have seen and propagated the warning message. Based on empirical results of time to propagate code over one hop, Freshet estimates when the hop count is sufficiently large that energy savings are possible by stopping advertising and turning the node's antenna off. In this exposition, we will use the term a node going off to sleep to mean its antenna being turned off. If the node has some sensing task, for which it needs to stay awake, without communicating, it can continue to do so. On getting the hop count information, the node starts a timer for how long to cease advertisements and go to sleep.

Given that the sleeping will happen for a source to node distance beyond *h* hops, a node h_a hops away sleeps for time $t_{off}*(h_a-(h-1))$, where $h_a > h$ and t_{off} is the time for the three way handshake between two neighbor nodes. This parameter is estimated empirically in the sensor network test bed. The additive nature of this formula stems from the result from Deluge that the time to propagate a page is linear in the number of hops for a fixed object size [6]. With further accurate information about the topology, it may be possible for each node to estimate the timeout more accurately.

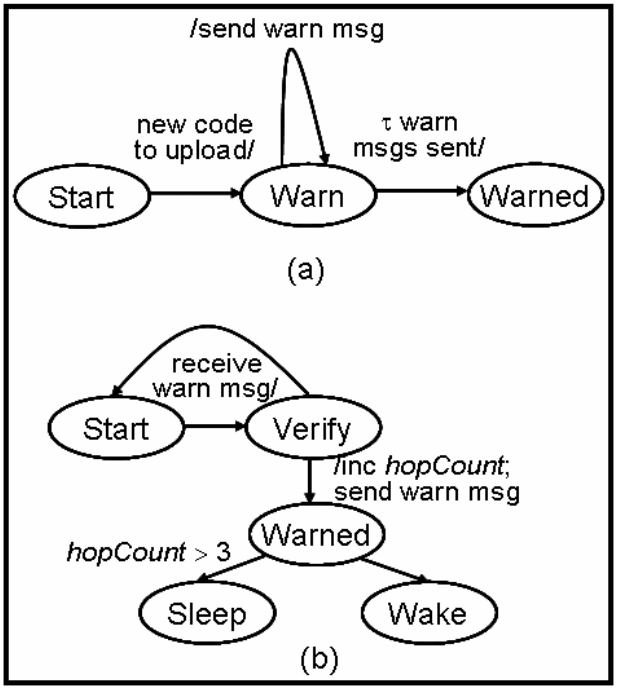


Figure 1. State machine in blitzkrieg phase. The labels on the edges are of the form *a*/*b* where *a* is the input that triggers the state transition and *b* is the output that is generated as part of the state transition.

We discuss in Section 4.1 an extension to Freshet where accurate location information is available.

The blitzkrieg phase causes each node to relay the warning message a fixed number of times, the redundancy being used to guard against losses. The blitzkrieg phase does not require any synchronization between the nodes and each node terminates its blitzkrieg phase when it has sent out the fixed number of warning messages. The state machine representations for an originator node and a general node in the blitzkrieg phase are shown in Figure 1(a) and (b) respectively. Figure 1(a) shows the process at the beginning of a code update to transmit warning messages. Once a node either hears newer code or a warning message from another source, it sends warning messages until it has sent and heard τ messages.

In Figure 1(b), we see that once a node has heard a warning message, it verifies that the metadata is an update to its current code image. If this is determined to be the case, the node starts sending out warning messages. Once finished, the node sleeps if it is more than 3 hops from code update, and stays awake otherwise. There are tradeoffs in determining how far away a node may be before it sleeps after the blitzkrieg phase.

Freshet uses 3 hops as the cutoff point because it balances energy savings with code proximity. Consider the nodes, say belonging to a set called A, that receive the blitzkrieg message directly from the originating node, call it node B. However, there may be other nodes, say belonging to a set called C, within range of the originator for which the blitzkrieg packet directly from the originator was dropped. But when the actual code transfer occurs, nodes in set C receive enough packets from node B to successfully download the code, as if they had originally been in set A. Since some nodes for each blitzkrieg message may fall into set C, there needs to be some

concession to incorporate these nodes. This is done by allowing a threshold for hop count for sleeping that is greater than 1. On the other hand, to allow for more energy savings Freshet would like to turn the radio off for as many nodes as possible. This factor pushes the value of the threshold lower. Empirically, 3 hops was determined as a suitable value; 2 typically was too restrictive on the nodes that could receive code updates and 4 typically saved less energy without significantly altering the download time.

As [17] shows, the major energy expenditure for the radio is the idle receive time and not the transmission energy level or number of messages sent. Therefore, Freshet seeks to turn off the radio between the blitzkrieg and the distribution phases. MNP in [7] turns off the radio of nodes which are not selected as senders of code (during their counterpart of the distribution phase), but does not address radio usage in the long time periods before and after code updates. Since a node can go to sleep between the time that code is injected into the network and when it arrives in the node's vicinity, a large network that needs to disseminate a large data object can save substantial amounts of energy in Freshet.

3.3 Distribution Phase

The distribution phase of Deluge achieves efficient and robust dissemination of code pages. Thus, Freshet leaves this phase unchanged and chooses to optimize aspects of Deluge not associated with the active distribution of code, while still maintaining the same performance. This phase is described in brief here for the sake of completeness.

The pseudo-code showing the operation of the distribution phase is shown in the Appendix in Figure 27. The state machine representation for a general node in the distribution phase is shown in Figure 2.

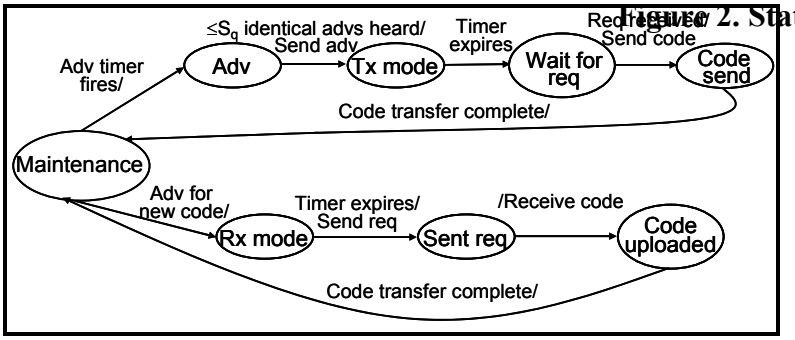


Figure 2. State machine in distribution phase

The distribution phase does not need any synchronization between the nodes. It begins once a node wakes up from the sleep induced by the warning message of the blitzkrieg phase, or, if it was determined that the node need not go to sleep, then right after the completion of the blitzkrieg phase. The distribution phase 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. Since a node only fills its pages in monotonically increasing order, it eliminates 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, subject to the monotonicity constraint mentioned above.

In addition to the advertisement suppression mentioned above, Freshet uses several mechanisms for message suppression. The first is sender selection. When a node needs new code, it designates the node to send the new code image. This sender is selected by the most recently heard advertisement and the other senders are thus quieted. The second mechanism is request suppression. When a node overhears a request for the same code it needs, then it suppresses its request, unless it does not receive the new code within some time interval.

3.4 Quiescent phase

A node enters the quiescent phase once code has been disseminated completely within the transmission range of the node. Thus, it no longer hears requests and it has itself acquired the complete code image. Since there will be no further code transfers for the immediate future, the node does not need to advertise at all. The two distinct scenarios that are to be handled in the quiescent phase are when a new node enters the network and when new code is injected into an existing network.

In Trickle [1], a scheme is proposed for sending an advertisement every so often to ensure that if a new node is added to the network, it is aware of the current code status. However, since the quiescent phase is typically the most long-lasting phase, Freshet optimizes the energy consumption further by switching to a complete pull-based mechanism to service new nodes. If any new node enters the network, it will advertise its old data and thus will alert the already present nodes that they need to start transmitting again. As it is difficult to decide deterministically when a node may safely shut off its radio, the quiescent phase operates by ensuring that all nodes in the network are awake at least half the time. Since this new node may

enter the network at any location and new code may be injected at any time, only a portion of the network can sleep and the nodes that sleep must probabilistically ensure that the network will still respond to any new events. The means of accomplishing this is through recording how many neighbors, b_n, are within each node's vicinity. Consider a time slot of length τ. Each node listens for a period $\tau/2$ and then decides with probability $1-1/b_n$ that it should sleep for the next $\tau/2$ period. This design is a tradeoff between energy saving and responsiveness of the network to new code or new nodes. The decision to have the node sleep half the time was modeled off of Trickle's methodology of equally dividing the time period into a listen and a send window. Further experimentation is required to determine the optimal sleep period for a given network.

In the case where new code enters the network, nodes that are awake will propagate the warning message throughout. Therefore all nodes awake when this occurs will be prepared for the new update. However, the portion of the network that was sleeping may have problems being prepared for the next update. However, note that it is very unlikely that the node will miss the code update completely, as it will be awake for half the time. Consequently, it will either have heard the initial warning messages or be aware when the code reaches a few hops away, as the nodes that received warning messages will have awakened by then and be sending advertisements to the surrounding nodes.

Freshet can function in either a dynamic or a static network. The dynamic nature may be a result of failures, which will cause new routes to be discovered that Freshet will use in the propagation of code. For a mobile network, two cases have to be considered. One is the node which wishes to upgrade its code is moving, in which case the node disregards any network topology information obtained earlier and stays awake for the code transfer. Since the energy expended due to motion is significantly higher than that due to listening energy, this appears to

be a reasonable choice. An exception would be the case where the transportation was provided by a host, such as a walking person. This case would rarely be practical since a fairly large network would require either many persons to move all the nodes in a synchronized manner and the network's environment is often not suitable to human intervention. The second case is the originator is mobile. It executes the blitzkrieg phase twice – once at the old location canceling the hop count information and again at the new location to update the nodes with the correct hop information. Note that a mobile originator that cancels its original warning message could negatively affect the network performance. Nodes more than *h* hops from the originator, where *h* is the number of hops beyond which nodes sleep before entering the distribution phase, may be asleep when the originator sends out the cancellation message. In this case, if the new location of the originator is closer to those nodes, then the distribution of code will be delayed. The speed of the mobile originator would also affect network performance. A slow-moving originator that moves across a large portion of the network may reach its new location at a much later time, such that the network has re-entered the quiescent phase while waiting for the code update. In this case, there would be a small impact on performance since it would be akin to have a newer version of the code appear in a neighborhood when nodes are in the quiescent phase. On the other hand, a fast-moving originator could reach parts of the network that are sleeping. In this case, code upload would take longer waiting for the nodes to awake.

The pseudo-code for the quiescent phase is shown in the Appendix in Figure 28.

4 Extensions to Freshet

In this section, we discuss three additional features of Freshet, augmenting the basic design.

4.1 Freshet with Location Information

In this extension, we equip Freshet with precise location information for the nodes. In the basic version of Freshet, the only network information available to a node is the number of hops it is

distant from the source of the data. However, due to the variability of the wireless channel, not all hops are made equal. Simply put, a single hop channel between two nodes 50 ft apart may be substantially more unreliable than one between nodes 10 ft apart. The unit time to transfer code of multiple packets over the lower reliability link will be higher since all the packets of a page must be received for the page to be successful. The wireless channel characteristic is dynamic and therefore, the number of hops traversed by the warning message may not be representative of the number of hops traversed during the actual code upload. The hypothesis is that given richer information on network topology, a node may improve its knowledge of how far it is from an injected code image and thus improve the estimate of the time to sleep. In the basic version, the design is motivated by energy savings and therefore each node picks a conservatively high value of time to sleep, giving an operating point of low energy consumption and high delay. The information that we choose for refining this estimate is the location information. Each node is aware of its location and disseminates this with the warning message during the blitzkrieg phase.

In this system model, each node either knows its own location with special hardware, such as a GPS receiver, or may obtain it through a network protocol using nodes with location information, such as our protocol in [18]. The mapping of distance from code source to delay can be made through analysis, provided the constituent delays can be represented using closed form formulae. In the case of our experimental testbed, this appears not to be the case due to the nature of the MAC layer protocol called B-MAC [19], which is a variant of the 802.11 CSMA/CA MAC protocol. The determination of the time to propagate code is thus from a pre-determined equation based on empirical results. The empirical result depends on the size and density of nodes in the network and thus, this is additional information pre-loaded into each node. In the current design, the nodes make a lower bound estimate on the code propagation time to optimize

for latency. Figure 3 and Figure 4 demonstrate the correlation between the distance from the code source and the time to disseminate code using the TOSSIM simulations. Both figures represent the time for complete download of the first page.

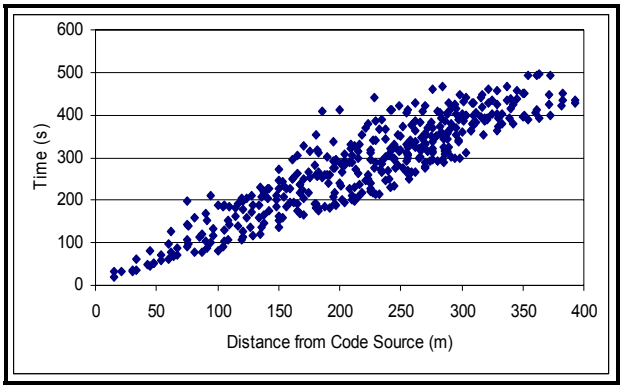


Figure 3. Time for dissemination of one page for a 400 node network

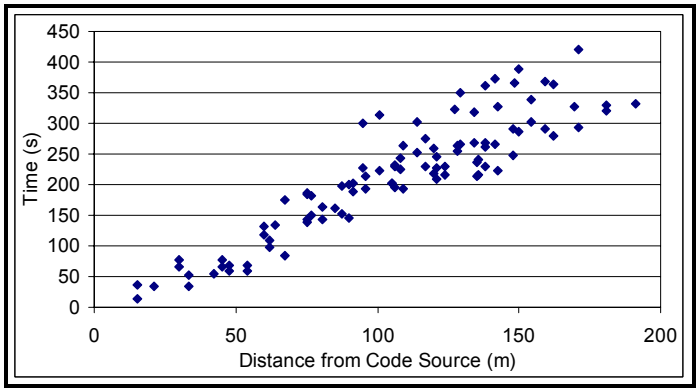


Figure 4. Time for dissemination of one page for a 100 node network

These figures are generated by running Freshet without any sleeping nodes and thus give an estimate of the best case performance, i.e., lowest delay for code propagation. The behavior of this characteristic is approximately linear with distance (correlation of line is 0.83 and 0.82 respectively for the 400 and 100 node networks), so we can approximate the time for a node to sleep through linear regression analysis for a given network size.

4.2 Multiple Page Transfer

The second extension is to optimize the number of control messages using knowledge of the pattern of code dissemination. The authors of [6] show that even with aggressive advertisement suppression in Deluge 18% of all packets are control packets. In particular, when a new code image enters the network, handshakes for each page – the cycle of advertisement, request, and code – delay progress in pushing code through the network. We target this source of overhead in Freshet to increase the utilization of the channel bandwidth. The underlying intuition is that if a large fraction of the neighbors of a node need several pages, the node can send these pages without repeated iterations of the handshake cycle. We call this mode the *multi-page mode*.

This trigger for the multi-page mode is reached by listening to advertisement messages. When a code sender only hears advertisements for older code images, then this sender is aware that its new update will be needed by all nodes within its immediate range. In this case, it is beneficial to optimize channel use by sending the multiple requested pages as quickly as possible without sending advertisements for each individual page. A node needing code assumes that the sender will send the appropriate pages without continuing to request those pages. If a node doesn't successfully receive all the packets of a page, then it sends a request for a retransmission. This is the only source of control packets in the multi-page mode. Following a given wait period, the sender transmits the next page without having had to advertise it, and without having had it requested. This reduces the code upload delay and improves channel utilization.

Figure 5 shows the state transition diagram for this handshake scenario – the upper half corresponds to the sending node and the lower half to the receiving node.

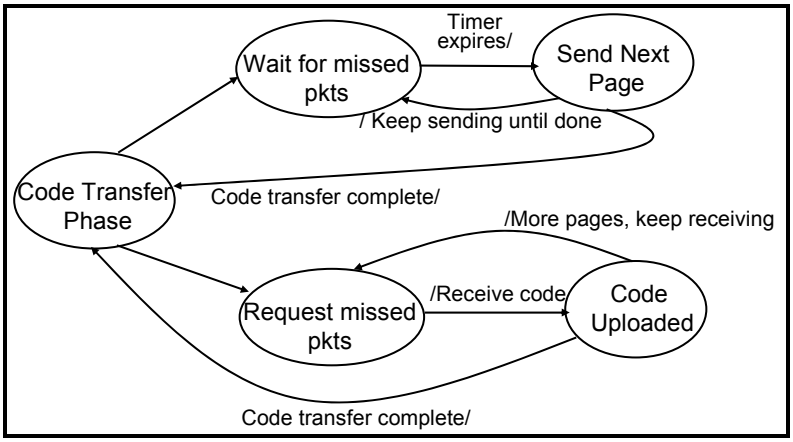


Figure 5. State transition diagram for multi-page mode

4.3 Multiple Originators

This component of the design of Freshet deals with situations where a network may have multiple identical code sources in different locations. In many cases with a deployed sensor network it is hard to access nodes inside the mesh of the network, but easy to access the outside

edges of the network. A user may deploy additional sources with the goal of reducing the time to propagate code through the network. Recollect that the term originator refers to one of the original sources that initiated the code propagation.

In Freshet, the use of multiple data originators would effectively partition the network into smaller portions. We propose a scheme to distribute pages out of order to improve dissemination in the network as a whole. Through out of order dissemination of pages it is possible that when pages distributed from different originators meet, they may fill in the "gaps" in each node's code image. This allows us to create fresh sources from which code can be disseminated. In this design, it is fundamentally important to design negotiation scheme so that collisions between multiple nodes trying to push code can be handled.

Thus, we propose the concept of node *parity*, where the parity of a node is determined by which set of pages it chooses to disseminate first when it already knows that there are other originators in the network sending pages with different parity. In particular, Freshet has *numSrc* originators sending code of size p_{max} pages into the network. For a given originator s_j, said to have parity j ($0 \leq j < numSrc$), it will first send out pages numbered i such that $i \bmod numSrc = j$. After distributing these $p_{max}/numSrc$ pages, it will then distribute pages numbered i such that $i \bmod numSrc = j-1, j-2, \ldots, 0$ and then $numSrc-1, \ldots, j+1$. It is assumed that the deployment of the originators is done with some thought – they are relatively evenly spread and are assigned non overlapping parities.

The next problem is how to resolve conflicts between nodes with pages of different parity. For a node with an incomplete image there is the concept of *cycles*, one for each parity in the network, with the node switching through the different cycles. Consider Figure 6 which depicts node behavior in a network with two parities. It goes through an even cycle and an odd cycle.

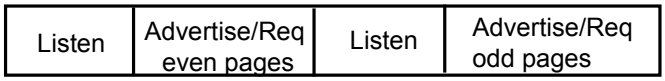


Figure 6. Cycles for 2 originators

Each cycle has one *slot* for listening and one for advertising and requesting. The cycle is dedicated to the particular parity when activity pertaining to both parities is happening around the node. However, if the node hears *a* consecutive advertisements of one parity, where *a* is a user-defined parameter, then it will use all available cycles for that parity. This is to ensure that cycles are not idled for pages of a given parity that are still far off from a node. As in Deluge, pages may only be downloaded sequentially within that parity. Thus, with two parities, the motes must download page 5 before page 7.

An optimization in Freshet for interleaved pages is that if a node's radio is idle in a given cycle and data is available, the node will utilize the cycle to get the data. What *is* sacrosanct is that a node does not transmit meta-data outside the turn. This is important to prevent the protocol from thrashing in which only meta-data exchanges happen and the network's throughput goes to zero.

5 Coordination with user application's sleep/awake scheme

We have to be aware that Freshet 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 power consumption due to idle listening and thus to lengthen the lifetime of the node. In our discussion of Freshet so far, we have assumed that Freshet can put the node to sleep according to its own calculation, disregarding the fact that the user application may require the node to be in the awake state to perform its tasks during this interval. In reality, Freshet cannot make this

unilateral decision. Here we discuss the simple change to Freshet so that it can co-exist with low duty cycle user applications.

We address this issue by mandating that Freshet is informed of the sleep-awake schedule of the user application. The condition for putting a node to the sleep state becomes:

1. User application puts the node to the sleep state AND
2. Freshet puts the node to the sleep state either between the blitzkrieg and the distribution phases (according to Section 3.2) or during the quiescent phase (according to Section 3.4).

The pseudo-code for managing the sleep/awake schedule is shown in Figure 8. Freshet uses 4 timers: *Freshet_sleep_timer*, *Freshet_awake_timer*, *UA_sleep_timer*, and *UA_awake_timer*. If these are hardware timers (there are 4 for the Mica2 platform), then the microcontroller can also be in the sleep state when the node is put to sleep. UA_sleep_interval and UA_awake_interval are the user application defined awake and sleep intervals respectively.

```
1. UA_awake_timer.fired()
        UA_sleep=FALSE
        UA_sleep_timer.start(UA_awake_interval)
2.  UA_sleep_timer.fired()
        UA_sleep=TRUE
        UA_awake_timer.start(UA_sleep_interval)
        Put the node to sleep
3. Freshet decides to put a node to the sleep state (either between blitzkrieg phase and distribution phase or
   during quiescent phase)
        t1 = Duration for which Freshet decides to put the node to the sleep state
        if(UA_sleep= =FALSE)
                t2=UA_awake_interval-(time elapsed since UA_awake_timer fired)
                Freshet_sleep_timer.start(t2)
        else
                Freshet_sleep_timer.start(0)
4. Freshet_sleep_timer.fired()
        t3=UA_sleep_interval-(time elapsed since UA_sleep_timer fired)
        if(t3<t1)
                Freshet_awake_timer.start(t3)
        else
                Freshet_awake_timer.start(t1)
        Put the node to sleep
5. Freshet_awake_timer.fired()
Wake up the node
```

Figure 7. Pseudo-code for Freshet co-existing with an application that has its intrinsic sleep-wake schedule for energy conservation

6 Analysis

6.1 Analysis 1: Number of redundant advertisements

First we analyze the number of redundant advertisements that are needed to achieve a given reliability of reaching a node in the network which is *relatively* isolated. This is defined as the *reliability of the code update protocol*. Let the number of nodes in the network be n, the size of the sensor field be A, and the radius of transmission be r_0. We assume for the analysis that the nodes are uniformly distributed in the sensor field. The density of the sensor field is $\rho = \frac{n}{A}$ and the average number of nodes in the transmission range of a given node is $\lambda = \pi r_0^2 \rho$. The probability that the number of neighbors of a node (d) is n_0 is given by a Poisson distribution. $P(d = n_0) = \frac{\lambda^{n_0}}{n_0!} e^{-\lambda}$, n_0=1,…,n, assuming $n >> n_0$. Let us consider an arbitrarily isolated node α that is a fraction τ of the SD away from the mean. Thus, the number of neighbors of the isolated node is $b_\alpha = E(d) - \tau S(d) = \sqrt{\lambda}(\sqrt{\lambda} - \tau)$, $\tau < 1$.

Now, consider the probability of successful transmission of an advertisement from one of the neighbors of α to node α. Note that we only need to consider a successful transmission of the advertisement and not the subsequent request and code packets since if node α is made aware of the presence of new code, it will continue to request arbitrarily long till successful transmission of the code is achieved. Of course, realistically collisions will cease on the channel to node α and the transmission will be successful within a few attempts. In order to estimate the probability of successful transmission of the advertisement, we use the analysis of the 802.11 CSMA/CA protocol given in [20]. Note that the Berkeley Mica motes do not use 802.11; the default MAC layer is B-MAC. Like 802.11 analyzed here, B-MAC uses CSMA/CA, but it does not use slots or

binary exponential backoff as 802.11 does. Instead, it uses continuous time and uniform backoff as the default (the user application may override the default though). Thus, our analysis is an approximation of the performance that can be seen with B-MAC. However, to the best of our knowledge, there does not exist an exact analysis of the B-MAC protocol.

For our analysis, binary exponential backoff is being used with minimum size of the contention window $CW_{min} = 2^{m}W$ and the maximum size $CW_{max} = 2^{m'}W$. We assume that any contention for the wireless channel comes from the neighbors of node α. The number of retries by a given node for transmitting the advertisement is then $M = m'-m+1$. The probability of successful transmission in one time slot is $P_s = P_{tr}P_{s|1}$, where P_{tr} is the probability that there is transmission and $P_{s|1}$ is the probability of successful transmission in a slot, given there is a transmission. We obtain using equations (10) and (11) in [20], $P_{tr} = 1-(1-P_t)^{b\alpha}$ and $P_{s|1} = \frac{b_\alpha \mathrm{P_t}(1-\mathrm{P_t})^{b_\alpha - 1}}{1-(1-\mathrm{P_t})^{b_\alpha - 1}}$, where P_t is the probability that a station chooses to transmit at a randomly chosen slot time and is given by equation (7).

Therefore, the probability of successful transmission $P_S = 1-(1-P_s)^M$, assuming that the probability in each time slot is *i.i.d*. Therefore the probability of success of at least one advertisement from among the r sent by a node i which is a neighbor of node α is $P_{S,i} = 1-(1-P_S)^{\mathrm{r}}$. Therefore the probability of success of at least one advertisement reaching the node α, i.e., by definition the reliability of the protocol, is $R = 1-(1-P_{S,i})^{\mathrm{b}\alpha}$. This can be made arbitrarily close to 1 by increasing the value of r and asymptotically goes to 1 as $r \to \infty$.

The analytical results are plotted in Figure 8 and Figure 9 for $n = 15\times15$, $A = 200\times200$, $CW_{min} = 16$, $CW_{max} = 1024$ from the 802.11 standard for FHSS Physical layer, Transmission power = -20dBm, and minimum Receive power = -85dBm giving $r_0 = 39.0937$ m (for the Mica motes).

Figure 8 shows the non intuitive result that the number of retries is not monotonically increasing with increasing τ. For higher values of P_t, the increased contention due to the number of neighbors of the isolated node causes the number of retries to decrease with τ to a minimum before increasing. Figure 9 shows that the reliability asymptotically approaches 1 which puts the reliability claim of Freshet on the same ground as that of other epidemic based protocols.

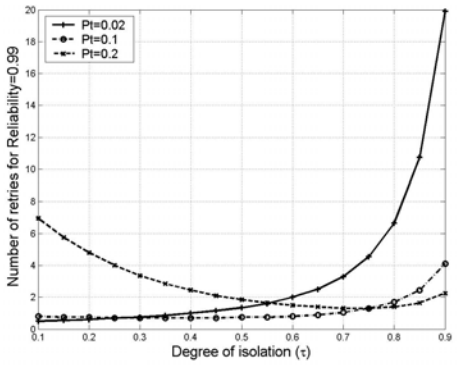


Figure 8: Variation of no. of retries to reach 99% reliability for an isolated node

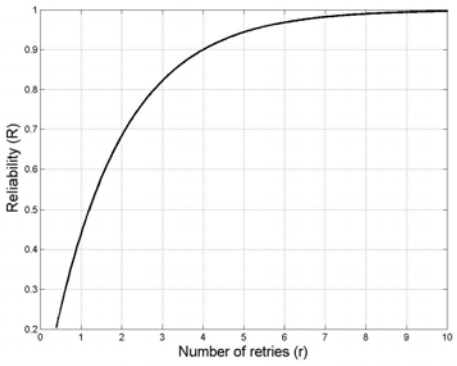


Figure 9: Variation of reliability with no. of retries for an isolated node (τ=0.9)

6.2 Analysis 2: Time between blitzkrieg and distribution phases

Next, we analyze the separation in time between the blitzkrieg and the distribution phases. For a node one hop away from the originator of the new code, this time interval is the time for a single round of a three way handshake. Assuming perfect pipelining of the single page of the code, the time interval $T_{delay,h}$ between the blitzkrieg and the distribution phases for a node h hops away from the originator of the new code is $T_{delay,h} = h.T_{round}$ where T_{round} is the time for a single round of the three way handshake. T_{round} consists of following components:

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

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

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

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

The expected number of transmissions for a given packet is

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

T_{adv} can be approximated as follows:

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

where t_l is the approximate time interval between two advertisements. Note that reprogramming protocols like Deluge divide 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 for 802.11 and no closed form solutions exist. The curve shown in [21] indicates that for the region of interest (low contention) the delay is approximately proportional to the square of the number of contending nodes. Here G is the proportionality constant and X is the number of contending nodes. T_x is the transmission time for a single packet. T_{proc} is the processing time required by a node after receiving the packet.

T_{req} can be calculated as follows:

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

where $E[N_{reqs}]$ 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:

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

where N is the number of packets in a page.

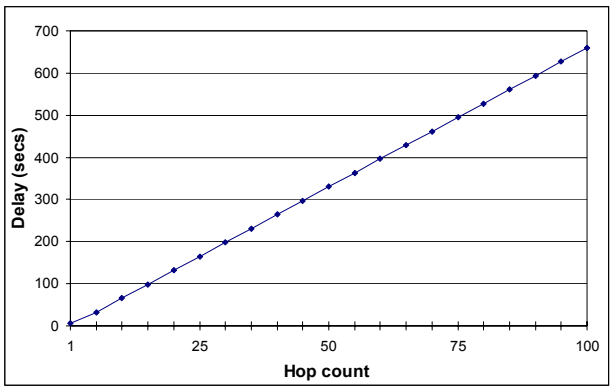


Figure 10 Time between blitzkrieg and distribution phases

Figure 10 shows the time interval between the blitzkrieg and the distribution phases as a function of hop count for a grid network with δ=10 ft separation between adjacent nodes. Probability of successful transmission of a packet P_s is taken as 0.9. For Deluge, t_l=2 seconds, t_r=0.5 seconds and N=48 packets.

From [6], we take $E[N_{reqs}]$=5.4. For mica2 node, transmission rate is 19.2 Kbps and hence T_x=0.015 seconds. We take T_{proc}=0.001 seconds. To calculate MAC delay GX^2, we take G=1 [13]. For a given node, the number of contending nodes varies with the location of the node in the network. For example, for the grid network, the nodes along the diagonal of the grid have higher number of contending nodes while those at the periphery have less contending nodes. We assume that the network is large and hence the average number of contending nodes is $9/4\delta^2$ (eliminating boundary effects) and the number of contending nodes is $9/4\delta^2 \times \pi r^2$ where r is the transmission range. The interference range of a node may be different from its transmission range. The difference can be easily accommodated in our analysis by replacing the

communication range with the given interference range. Figure 10 shows that the time between the blitzkrieg and the distribution phases is quite large for the nodes distant from the originator of the code. Under Freshet the nodes can sleep for this duration, and thus Freshet can conserve the energy for network reprogramming which increases with the network size.

6.3 Analysis 3: Effect of hop estimation on code propagation

In this analysis we will inspect the effect of hop estimation on saving energy and delaying download of a code update. Let us assume a square network of arbitrarily large size. The code source is node A, and we will investigate the propagation time to a node B h hops away from A.

Let the expected propagation time of one page between two nodes one hop away be D and the variance be V. The propagation delay between any two nodes is assumed independent of that between the next set of nodes. Let X be the random variable for the time to propagate one page from node A to node B. Using the central limit theorem, X follows a Normal distribution with mean $D_{agg} = h*D$ and variance $\sigma^2 = h*V$, for reasonably large h, say greater than 10. Given these parameters, we wish to select a sleep period for node B that ensures high energy savings and guarantees with high probability that the code update reaches node B while B is awake. Therefore we wish to select the time to sleep, T_{sleep}, as some value $D_{agg}+f*\sigma$, where f is in the set of real numbers, greater or less than zero. Since X is normally distributed, we can calculate the probability for a given f that B will be awake when it sees the code update; we can also calculate the expected energy savings for a given value of f. Since Deluge does not turn off its radio at all, the energy savings of Freshet corresponds to the entire time that the radio is turned off. Therefore, the expected energy savings for parameter f is (3 V)(7.03 mA)($D_{agg}+f\ \sigma$) (using parameters for the Mica2 mote). Assuming that D is 50 s and V is 225 s^2 (reasonable values as seen from the experiments – the high value of D is explained by the fact that each page has 48 packets, each of which needs to be received at the end of the link), this expression is graphed in

Figure 11 for h = 30. This figure shows the energy savings increasing linearly with f. However, there is a significant tradeoff for high f values. For instance, at f=0 there is 0.5 probability that node B will be asleep when the code update reaches it. This naturally seems problematic and will prevent a fast dissemination of the update.

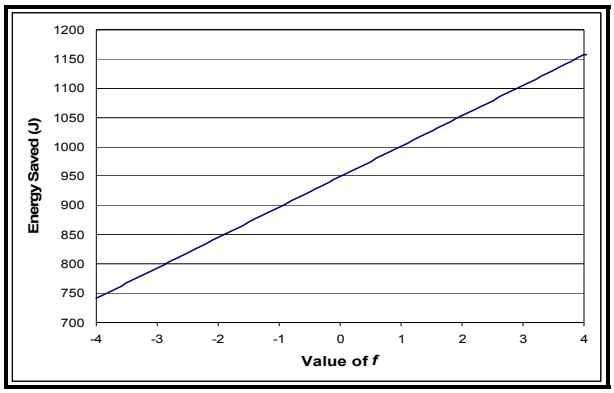


Figure 11. Energy savings with changing values of *f*

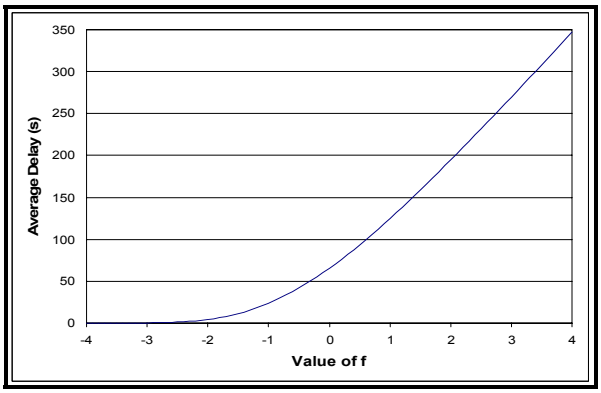


Figure 12. Average delay from sleeping in seconds for varying values of *f*

To determine the expected additional delay due to sleeping (conditional expectation, conditioned on the fact that there is additional delay due to sleeping), we subtract from the sleeping time, the expected time when the code reaches the node. For a given f the expected delay will be

$$E[Delay] = \left(D_{agg} + f\sigma\right) - E\left[\overline{X} \mid x \le D_{agg} + f\sigma\right] = \left(D_{agg} + f\sigma\right) - \frac{\int_{-\infty}^{D_{agg}+f\sigma} x * e^{-\left(x - D_{agg}\right)^2 / 2\sigma^2} dx}{\int_{-\infty}^{D_{agg}+f\sigma} e^{-\left(x - D_{agg}\right)^2 / 2\sigma^2} dx} \quad (1)$$

This expression is evaluated for f from -4 to 4 and is shown in Figure 12. It increases super-linearly with increasing f.

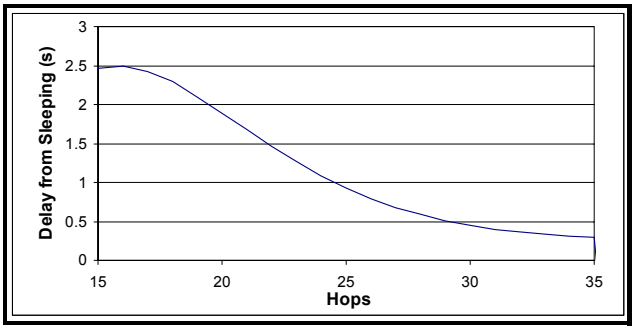
We extend our analysis to see what the effect is when the network experiences multiple delays due to nodes sleeping when the code reaches them. Let us consider a square network and a node A as the code source and a set S of nodes equidistant from A. While nodes in S are sleeping, the

network is partitioned. Each set of nodes after S will be labeled $S+k$, where $k = 1,\ldots,\infty$ is the number of hops between S and the set $S+k$. We again assume that there is no additional delay due to sleeping[2] at S due to nodes closer to A than S sleeping.

There are two cases to be considered for analyzing the delay of the set of nodes $S+n$ – the case where there is no prior sleeping and the case where there is prior sleeping. Let the total sleep delay at $S+n$ be represented by $R[S+n]$ and $D(S+n)$ be the expected value of delay due to sleep of $S+n$ under the condition that there is no delay due to sleeping prior to $S+n$. Let P_{asleep} represent the probability that $x \leq D_{\text{agg}}+f\sigma$ at a given node $S+i$. Therefore, probability that all nodes prior to $S+n$ are awake when they receive the code update is $(1-P_{\text{asleep}})^{n-1}$. For small enough n, P_{asleep} can reasonably be taken to remain constant since the time to sleep is proportional to the number of hops. Thus, the expected delay at $S+n$ given that there is no prior sleeping is $D(S+n)*(1- P_{\text{asleep}})^{n}$, where $D(S+n)$ is from equation (1) but with the modification to D_{agg} and σ according to the number of hops. The second component is the delay due to previous nodes. The delay at node S is $R[S]=D(S)$. The delay at nodes $S+1$ is broken into two cases – one where S is awake and another where S is asleep, giving the expectation expression $P_{\text{asleep}}*D(S+1)*(1-P_{\text{asleep}})+ P_{\text{asleep}}*P_{\text{asleep}}*X$. X is the expected delay due to sleeping at $S+1$ given sleeping at S. The sleeping delay at S is $R[S]$, but this sleep is time that $S+1$ may still sleep without any sleeping delay incurred. Therefore, the quantity X is the difference between the expected sleep at $S+1$ and the total sleep at S = $D(S+1)P_{asleep}-R[S]$. To force X to be positive, we define X = $\max(D(S+1)P_{asleep}-R[S], 0)$. Extending this analysis to nodes $S+n$, X becomes the difference between $D(S+n)$ and the sum of all $R[S+i]$ from $i=0$ to $n-1$. $R[S+n]$ becomes (1-

[2] Henceforth in the discussion, we will abbreviate additional delay due to sleeping by simply delay, where there is no scope for confusion. The implicit understanding is that normal delays due to propagation will be added to get the total delay.

$P_{\text{asleep}})^n * P_{asleep} * \text{D}(S+n) + (1-(1-P_{\text{asleep}})^n) * (\text{D}(S+\text{n}) P_{asleep} - \sum R[S+i])$, which simplifies to $(P_{\text{asleep}} * \text{D}(S+n) - (1-(1- P_{\text{asleep}})^n) * \sum R[S+i])$.



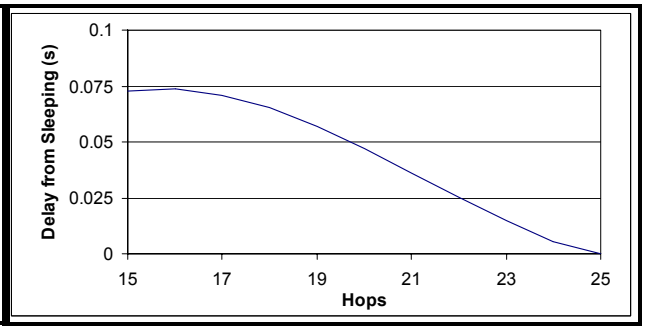


Figure 13. Sleeping delay with # hops *f*=-1.0 **Figure 14. Sleeping delay with # hops *f*=-2.0**

Figure 13 and Figure 14 show the delay from sleeping as hops from the source are increased, with the set *S* at hop 15 for *f*=-1 and *f*=-2. It is noteworthy that as the number of hops increases, the delay due to excess sleeping will disappear. Thus beyond a certain number of hops (35 for *f*=-1, 25 for *f*=-2), the nodes will always be awake when the code arrives. The accumulation of delay shows that if the code reaches some part of the network that is asleep and must wait, the delay due to sleeping incurred at that point has progressively less effect as the code goes away from that part of the network.

7 Experiments and Results

We simulate Deluge (from TinyOS release 1.1.11) and Freshet (built on top of this release of Deluge) using TOSSIM. While TOSSIM does not imitate hardware precisely, its purpose in these experiments is to compare Deluge's performance to that of Freshet in larger networks. Any changes in code dissemination time or behavior a real-world environment due to the approximations of the simulator would apply equally to Deluge and Freshet. This argument for interpreting trends and comparative evaluation using TOSSIM has been made by countless researchers in the field, such as in [6, 7]. The TOSSIM code runs directly on hardware and closely mimics the trend in the network behavior, though the measurements do not give accurate

absolute numbers. Work presented in [25], [26], and [27] discusses some of the more important differences between simulators and real-world wireless sensor network implementations. The gains of Freshet are evident for network sizes of the order of tens to hundreds of nodes and therefore TOSSIM rather than the actual motes were used for the results showing the comparative gains of Freshet. This approach is valid because of the accuracy of the simulation infrastructure and has been used by other researchers [6, 7]. The code is fragmented into pages each consisting of 48 packets of 36 bytes. The nodes are arranged in a rectangular grid with constant 15 ft. spacing between adjacent grid points. A square placement of nodes on the grid is used to give N×N nodes, where N is varied for the experiments. Henceforth, the term "N nodes square" will imply a total of N^2 nodes in the network. The amount of sleep time for a node *h* hops away from the warning message is *8(h-1)* for $h \geq 4$. This equation was found empirically and generally yielded adequate responsiveness in the network while still guaranteeing some period of sleeping for nodes far from the source of the code. For experiments with location information, we independently found the best fit for each network size. This helped create the most reasonable estimate of code propagation speed in a given network. The BER was set for each link through use of the TinyOS LossyBuilder tool. We used the default communication range of 50 ft for the simulations. We acknowledge that this loss model is specific to the empirical setting of TOSSIM's LossyBuilder and is used as a reference model for comparison with Deluge. The authors of Deluge had also used this model for link losses.

TOSSIM does not have built in simulation for energy computation, nor does it have a radio model with power management features. To work around this problem, we used PowerTOSSIM [17] to track energy usage. For energy consumption we used the Mica-2 hardware model with the parameters as in Table 7.1. As shown in [6], the completion time in Deluge scales linearly

with object size. Through our Freshet experiments we found that energy use followed a linear increase with object size as well, and hence we do not discuss results with varying object size.

7.1.1 Single Originator Results

We run our first set of experiments with code image consisting of 5 pages in networks of sizes of 6-20 motes square. The simulations are run 3 times for each network size. They are started with all the nodes being active, and at 10 seconds into the simulation the originator starts transmitting the code pages. The simulations are run until all the nodes receive all the pages, which is the time presented in the results as the time for code upload.

Table 7.1: Energy model used for experiments

Radio idle or receive	7.03 mA	EEPROM Write current	18.4 mA
Radio transmission (max transmit only)	21.5 mA	EEPROM Write time	12.9 ms
CPU Active, Idle	8.0 mA, 3.2 mA	EEPROM Read current	6.2 mA
Radio sleep	1 µA	EEPROM Read time	565 µs

In all cases we are evaluating the radio energy usage of Deluge and Freshet. We also track the CPU energy usage and energy from EEPROM writes and reads, but we found that the differences in this energy use due to these heads between Deluge and Freshet were negligible.

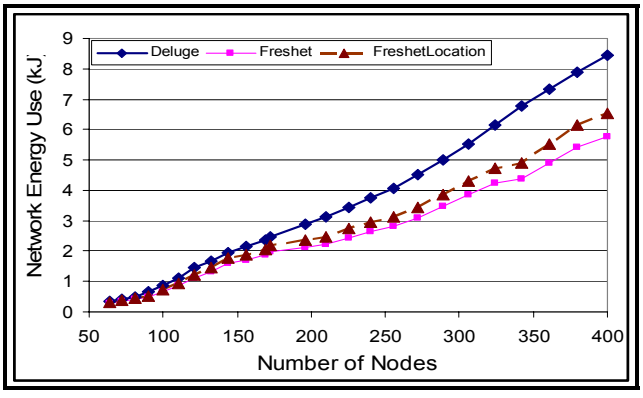


Figure 15 shows that as the number of nodes increases in the network Freshet saves more energy compared to Deluge. The energy gains of Freshet increase with network size since the energy spent per node is lower in Freshet.

Figure 15. Radio energy usage of the entire network for a given number of nodes

These plots give the total energy spent in the network and therefore scale based on the energy used per node. Clearly, a larger network uses more energy due to more nodes, but since there is

also more time for code to propagate, each node will need to spend more time waiting for code, which is used in Freshet for sleeping. This figure shows two main characteristics. First, the smaller networks use much less energy than the middle-sized networks. This is primarily due to the increase in the average hop distance between the originator and the nodes—in the 8x8 network the diameter of the network is 2-3 hops while in the 11x11 network it is 4-5 hops. Each hop increases download time and therefore increases energy use. However, as the network size continues to increase, the energy use begins to level off. We found that for up to a 10×10 network the propagation time is proportional to the product of the network diameter and the code size. Beyond that size it is proportional to the sum of the diameter and code size as shown in Figure 15 and in accordance with the result reported in [6]. Thus the total energy plot is approximately linear as the network size increases and the energy consumption per node levels off. Figure 15 also shows that Freshet with location information does not save as much energy as baseline Freshet, although it outperforms Deluge by a sizable margin. The location information "penalizes" Freshet because it causes nodes to turn their radios on earlier to minimize latency.

As far as time to completion, the location information grants greater granularity in estimating the time it will take code to reach the node. Let us consider nodes A, B, C, and D, where A is the code source, B is 15 feet from the code source, C is 30 feet, and D is 45 feet. The blitzkrieg phase working without location information propagates hop estimates through broadcast messages that if received properly will give the same hop count to node C and node B (and in some cases D). However, based on packet loss rates node C is less likely to receive that warning message at the same time as B, and therefore will probably be labeled as two hops from the sending node A. However, C is still within range of A when A starts transmitting the code update

and will likely receive *some* packets directly from A. Thus, the hop based model gives a higher estimate of time for code to reach a node compared to the accurate location based estimate.

Our simulations found that on average a data message propagates 19 feet in a network with 15 foot spacing between nodes. This implies that approximately once every three hops the data message propagates to one node 15 feet away and another 30 feet away. So in practical terms the situation outlined above occurs about 7 times in a linear network of 1 by 20 nodes, and naturally more frequently in a 20 by 20 network. This jumping beyond the nearest hop is less likely during the transmission of the warning message because of the higher level of congestion in the network. This leads to the result that the blitzkrieg phase overestimates the number of hops a node is away from the source.

As would be indicated by the design, the energy savings happen for two reasons. The nodes far from the originator node use the blitzkrieg phase to turn off their radios for the appropriate period of time before they must start transferring pages. The second reason is that nodes near the source that complete their code transfers first will have lower duty cycles for their radios as they enter the quiescent phase.

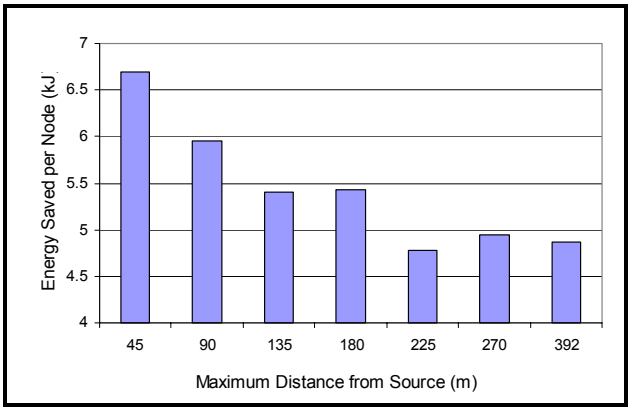


Figure 16. Average energy saved per node grouped by distance from code source

Figure 16 shows the average energy saved per node with distance from the code source for a 20x20 network. The energy saving is calculated as the difference between the idle radio power consumption and the node sleeping power consumption, multiplied by the time. The time is the time for the entire network to download the code completely.

The nodes closer to the originator are able to save energy through the quiescent phase by turning off their radios once they have acquired all of the code. Similarly, nodes far from the code source can save energy through the blitzkrieg phase but must still spend more time with their radios on to acquire the code updates.

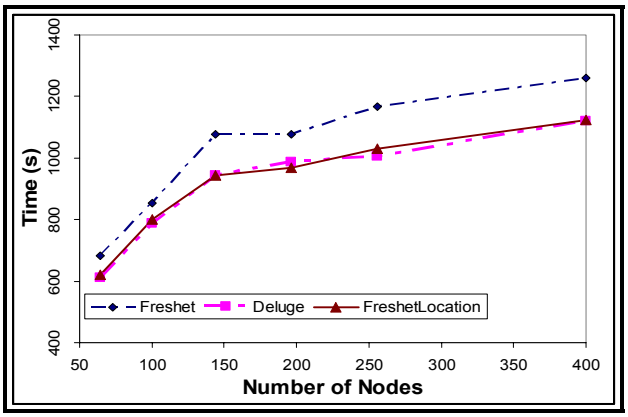


Figure 17. Time to complete code upload

Figure 17 shows the relative completion times for code upload of the three protocols. In all cases Deluge outperforms Freshet though Freshet with location information performs almost identically to Deluge. The location information helps Freshet minimize cases where the update reaches a sleeping node.

However, based on Figure 15 we see that Freshet uses less energy without the location information. The tradeoff indicates a design consideration – in cases where speed takes precedence, then it is better to have location information, but in cases where energy is more important, then location information is not necessary or the scheme that calculates the sleeping time based on location information has to be modified.

Figure 18 and Figure 19 demonstrate the profile of energy savings of the nodes in the network at two different time points of the code upload process. Figure 18 shows the distribution of node energy savings when 75% of the network has got the complete code. The energy savings at this point are due to the estimate of the time between the blitzkrieg and the distribution phases and sleeping for part of it. Figure 19 shows the same network 150 s after 92% of the network is completed. It is clear that a much larger percentage of the network has increased its energy savings in this time since the quiescent phase has set in.

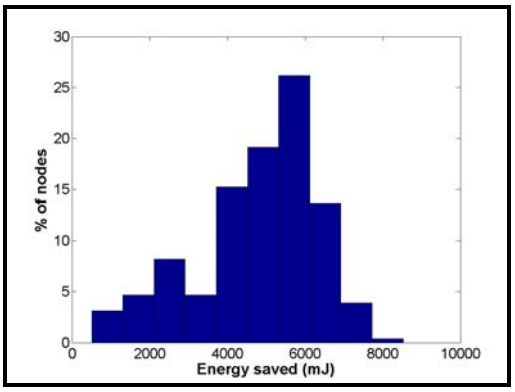


Figure 18. Profile of energy savings at 75% network completion

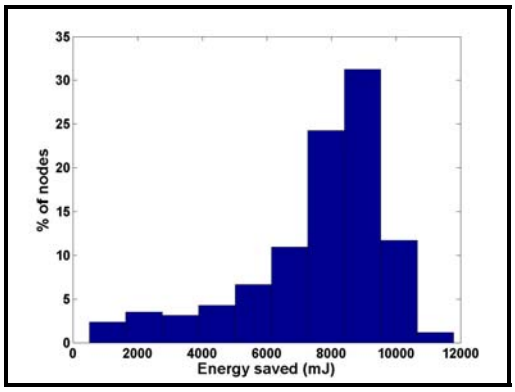


Figure 19. Profile of energy savings 150s after 92% of the network is completed

7.1.2 Multiple Originator Results

Our second set of experiments was run with two originators at the top left and bottom right corners and code size of 4 pages in networks consisting of 8 through 12 nodes square. We compare the performance of Deluge, with one and two originators and Freshet, also with one and two originators. In Freshet, one originator is set to prioritize distribution of even numbered pages and the other odd numbered pages.

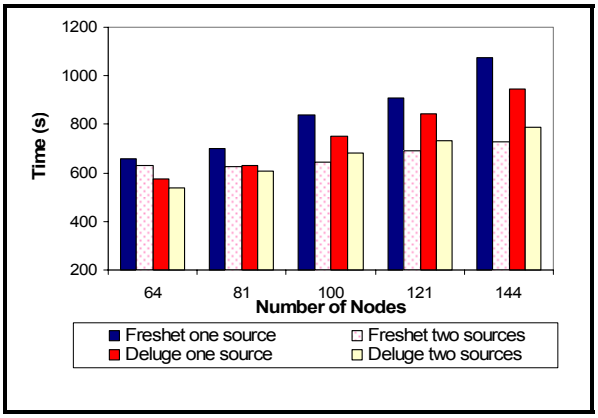


Figure 20. Time to completion of various distribution techniques. From left to right the techniques are Freshet one source, two sources, Deluge one source, and two sources

Figure 20 summarizes our results with the two Freshet bars to the left of the two Deluge bars. Multiple originators always improve performance in networks with ≥100 nodes. Specifically, when the originators are farther apart due to the larger network, the interleaving of pages in Freshet outperforms both Deluge with one or two originators.

This result occurs because of collisions in the code pages from the two originators for Freshet. This problem is the hidden terminal problem and limits the functionality in networks with less

than 100 nodes. For a sufficiently large network, however, page interleaving with proper contention resolution as in Freshet enables nodes near the middle of the network to complete downloading their code images earlier. They can then distribute code to others in the network.

7.2 Multiple Page Transfer

We conducted a series of experiments with different techniques for the multi-page transfer extension. The first experiment involved varying the number of packets sent per page, effectively increasing the size of the page sent per handshake and thereby reducing the control traffic. This network was a 2x10 network with uniform bit-error rates between adjacent nodes. The control parameter is the bit error rate (BER). This relationship is particularly important because it is the key in finding a proper page size. With a sufficiently reliable network, it is practical to send as many packets per page as possible. However with unreliable links, more control messages are used requesting packets lost in transmission. The advantage of limiting the page size is useful in networks with questionable reliability – a large page takes longer to download in a lossy network, increasing the time before the page can be propagated in a pipelined manner. In the experiment, packet size is constant at 36 bytes and each code image uploaded is 384 packets. The BER was varied till 1.5% and the effect on time to upload code measured for the two cases of 48 packets/page and 96 packets/page. Figure 21 shows our results. For smaller BER, transmitting the larger sized pages is advantageous due to the reduced amount of control traffic. Once the BER passes 1% we see a sharp increase in the time to transmit the code image in both cases. Once the BER gets sufficiently large (> 1.3%), the high loss rate of packets affects the performance of the larger-sized pages. Beyond BER 1.5%, the network did not function properly due to the high packet loss rate, which made simulations excessively long (1.5% BER ≡ 11% packet loss rate).

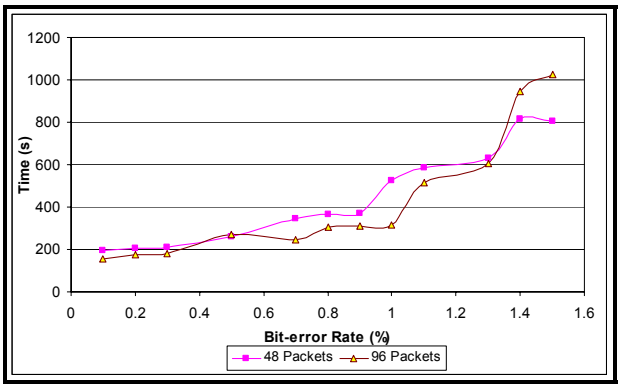


Figure 21. Effect of bit error rate on time for code upload with varying page sizes

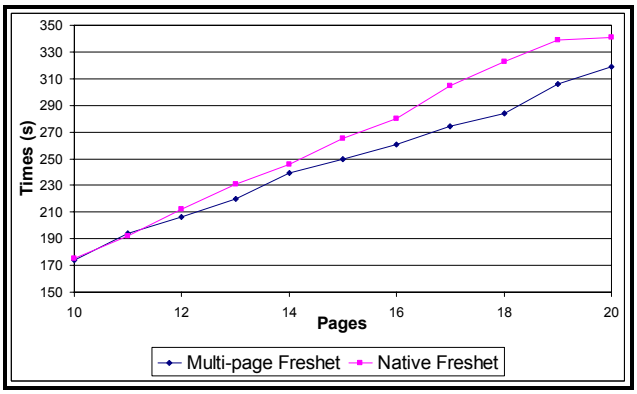


Figure 22. Comparison of baseline Freshet with multi-page Freshet

The second experiment sought to demonstrate the effect of sending multiple pages without the intervening handshake of advertisement and request between pages (Figure 22). The BER was configured through the TinyOS LossyBuilder utility, which generates network loss rates from the physical topology. Each page was the standard length of 48 packets. In the incremental page send mode, the node would continue to send pages till there was a request for retransmission due to packet loss. The experiment was conducted for getting the code uploaded into a node surrounded by 8 nodes on surrounding grid points each with the complete code. Visualize a 3×3 sub-grid with the middle node not having any part of the code. The number of pages in the code image was varied from 1 to 20. The results for less than 10 pages showed no noticeable difference. However, after 10 pages we noticed a significant difference between the standard Freshet and multi-page Freshet. This trend occurs because the extra control messages that normally occur in Freshet become sufficient to cause a delay in transmission of code.

7.3 Testbed Demonstration

As we discussed earlier, the advantages of Freshet over Deluge will be pronounced only for large networks with tens of hops. This would entail a testbed of several hundreds of nodes (note that as reported in [6], 75 nodes gave a 5 hop network). We do not have access to such a large testbed and therefore the purpose of the first set of the experiments on a 16 node testbed is to

demonstrate that in small networks Freshet performs comparably to Deluge. In the second set of experiments, we demonstrate the energy savings achieved by Freshet for linear networks having up to 15 hops.

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. For our experiments we used 2x2, 3x3 and 4x4 square grid networks having a distance of 5 ft between adjacent nodes in each row and column. This creates a network of diameter 3 hops by setting the transmission power level to 25 (of a range of 1-255). Lower values of the transmission power to increase the number of hops result in poor reliability and are therefore not used. Experiments of network reprogramming using Freshet are carried out by installing Freshet and same version of application code on all nodes in the network. A new code image 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 having all nodes install Deluge instead of Freshet.

Reliability of code upload is an important evaluation metric. 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*. 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 Freshet are 100% reliable, i.e. all nodes in the network download every byte of

the user application. So, in our experiments, we focus on time to reprogram the network and the energy consumed during reprogramming.

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 from each node. If the network has n nodes including the source node, the computer attached to the source node receives one t_0 and $(n-1)$ 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. Moreover, the time t_d is included for both Freshet and Deluge.

Figure 23 shows the average time taken by Freshet and Deluge to reprogram 2x2, 3x3 and 4x4 grid networks along with 99% confidence intervals. The reprogramming times shown in this figure are the averages taken over 10 experiments for each topology. For these small networks,

the reprogramming times of Deluge and Freshet should be equal. But we found that Freshet took 3.98% to 4.89% more time than Deluge to reprogram these networks because the size of Freshet is one page more than that of Deluge.

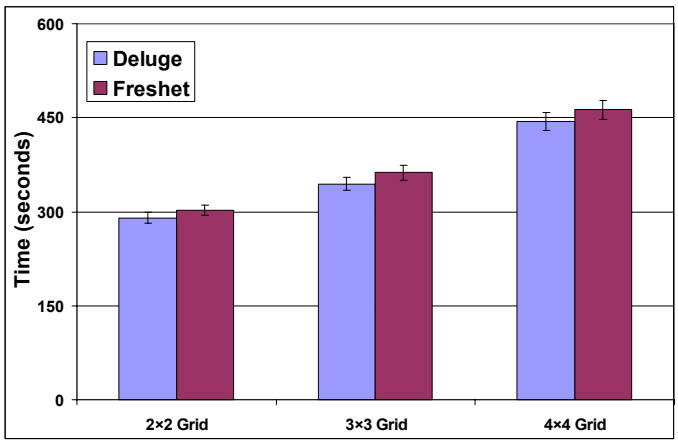


Figure 23. Reprogramming times of Freshet and Deluge

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). 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.

Both data packets and control packets (request and advertisement packets for Deluge and request, advertisement and warning packets for Freshet) are considered while calculating the number of bytes. In our experiments we found that Freshet transferred only 0.37% to 0.52% more number of bytes than Deluge. Although Freshet has one more packet type (warning

packets) than Deluge, its contribution is negligible because number of warning packets is insignificant compared to the other packet types. Also note that we have not counted the advertisement packets transmitted in the network during quiescent phase. If we consider them, the number of bytes transmitted in the network by Freshet will be smaller than that by Deluge.

As mentioned above, the advantage of Freshet over Deluge in terms of energy savings can be demonstrated only in larger 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 16 nodes. As shown in Figure 24 , a source node (node 0) situated at one end of the line disseminates code to all the nodes in the network. Let the nodes be arranged as shown in Figure 24 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 source node and the farthest node from the source node, we restrict the communication of a node *i* with node *(i-1)* and node *(i+1)* only. Each node logs the amount of time it sleeps between the blitzkrieg and distribution phases to its external Flash. This is used to calculate the energy savings compared to Deluge by using the formula: Savings = Voltage × (Current for idle radio + Current for idle CPU) × Sleeping time (as calculated from the experiments). The energy savings achieved by Freshet are shown in Figure 25. Note that this figure does not consider the energy saving because of the nodes sleeping in the quiescent phase. If we consider the energy savings in the quiescent phase also, the energy saving due to Freshet increases monotonically with time. Figure 25 shows that as the distance (number of hops) between the node and the source node increases, the energy saving increases linearly. This is because the amount of time each node sleeps increases linearly with hop-count. Due to our design of no sleeping for up to 4 hops, the energy saving only shows up beyond 4 hops. Note that we assume that both Deluge and Freshet consume the same transmission energy since the

numbers of packets transmitted in the network by Freshet and Deluge during reprogramming are almost equal (within 0.52%). It should be noted that the software control approach that we used to limit the communication of a node with its adjacent nodes achieves the maximum possible number of hops between the two end nodes in the line. However, it ignores the low level details, such as (1) sometime two nodes far away hear each other, (2) sometime links become asymmetric. Our goal in doing the experiment is to show through real testbeds that sleep time between the Blitzkrieg and the Distribution phases becomes significant only in larger hop networks. The testbed experiments for small grid networks (up to 4x4 grid) are done without the software control and hence all the low level details are reflected in the experimental results. An identical software control approach for maintaining the required topology in sensor networks has been used recently in [24]. Also the software control approach does not eliminate interference from nodes that are more than one hop away. This leads to more collisions than in a network where the nodes would be spread out farther. Since Freshet uses Deluge's approach of message suppression to combat such high density deployments, its effect on the overall performance is negligible.

Figure 24. Linear topology with nodes being reprogrammed using Freshet and Deluge starting with node 0 as the originator (*N*=1,2,…,16)

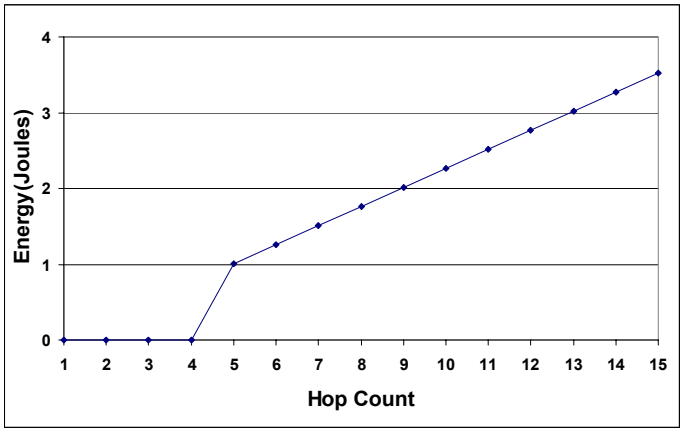


Figure 25. Energy saving achieved by Freshet over Deluge due to nodes sleeping between blitzkrieg and distribution phases

8 Conclusions

In this paper we have presented Freshet, a protocol for reliable code dissemination in a multi-hop sensor network. Freshet functions in three phases for each new code image – blitzkrieg, distribution, and quiescent. It aggressively conserves energy by putting nodes to sleep between the blitzkrieg and the distribution phases as well as the quiescent phase. Freshet introduces a scheme to disseminate code from multiple originators, use location information, and reduce control message overhead. Freshet is demonstrated using the TOSSIM simulator for the Berkeley motes and is found to be between 20-45% more efficient in energy compared to the Deluge protocol, while requiring about 10% more time for propagating the code.

In the future we plan to devise better strategies to predict the delay between the blitzkrieg and the distribution phases. We are looking at using better metrics to determine which node should be the local sender so that the maximum number of nodes can be satisfied. We are investigating the behavior of Freshet with faulty nodes and proposing appropriate increase in redundancy of the different messages that will make the network resilient to faults. The impact of different ratios of sleep to awake times on the performance of Freshet will be looked at in future work.

References

[1]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.

[2]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.

[3]P. Levis and D. Culler, "Mat\é: 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.

[4]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.

[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 L. Wang. "MNP: Multihop Network Reprogramming Service for Sensor Networks," at the 25th IEEE International Conference on Distributed Computing Systems, pp. 7-16, 2005.

[8]A. Tridgell and P. Mackerras, "Rsync," http://samba.anu.edu.au/rsync/documentation.html, 2005.

[9]Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," *Wireless Networks*, vol. 8, no. 2/3, pp. 153-167, 2002.

[10]S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, and J. F. Kurose, "Scalable reliable multicast using multiple multicast channels," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 294-310, 2000.

[11]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.

[12]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.

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

[14]S.-J. Park, R. Vedantham, R. Sivakumar, and I. F. Akyildiz, "A scalable approach for reliable downstream data delivery in wireless sensor networks," at the Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing, pp. 78-89, 2004.

[15]U. o. C. Berkeley, "TinyOS," *" At:* http://www.tinyos.net/.

[16]C. T. Inc., "Mote In-Network Programming User Reference," http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf, 2003.

[17]V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," at the Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA, pp. 188-200, 2004.

[18]N. Malhotra, M. Krasniewski, C. Yang, S. Bagchi, and W. Chappell, "Location Estimation in Ad Hoc Networks with Directional Antennas," *Proceedings. 25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, no., pp. 633-642, 2005.

[19]Y. Wei, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 3, pp. 493-506, 2004.

[20]G. Bianchi, "Performance analysis of the IEEE 802.11 distributed coordination function," *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 3, pp. 535-547, 2000.

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

[22]P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications." In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).

[23]Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," Network, IEEE 20(3) pp 48-55, May 2006.

[24]A. Kamra, J. Feldman, V. Sharma, and D. Rubenstein, "Growth codes - maximizing sensor network data persistence," at the Conference of the ACM Special Interest Group on Data Communications (SIGCOMM), pp. 255-266, 2006.

[25]G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic, "Impact of Radio Irregularity on Wireless Sensor Networks," MobiSYS 2004.

[26]J. Zhao and R. Govindan, "Understanding Packet Delivery Performance in Dense Wireless Sensor Networks," SenSys 2003.

[27]A. Woo, T. Tong, and D. Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks," SenSys 2003.

[28] R. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming," In Proceedings of the 26th Annual IEEE Conference on Computer Communications (INFOCOM), 2007.

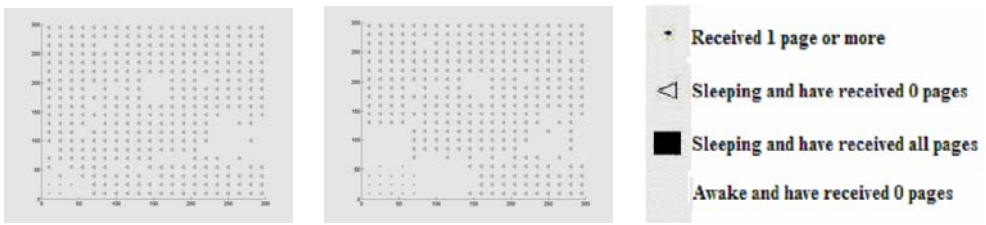
9 Appendix

9.1 Visualization of Network Behavior during Code Upload

The next part of our analysis centers on the network's behavior over time. Figure 26 shows the positions of sleeping nodes in 20×20 network as time progresses. The originator node is in the bottom left corner of the area. The small dots represent the nodes that have at least one page, the bigger dots (small solid triangles) represent nodes that are asleep, and the lack of any dot at a grid point represents a node that is awake but does not have a page yet.

Figure 26(a), (b), and (c) show that initially most of the network is asleep. In (d) most of the nodes have now turned their radios back on, and by (e) nearly all nodes in the network have at least one page. (f) shows the transfer of the code image to be complete, and in (g) we find that the nodes near the originator have now begun to sleep in the quiescent phase. By (h) a larger fraction of the network is sleeping in its quiescent phase.

These figures show that Freshet can reliably predict when to turn its motes' radios on and off, thereby saving substantial amounts of energy. In some cases we see that motes that are near those that have already obtained a complete page and should be ready for beginning the distribution phase, are actually asleep (some nodes to the right in (d)). However, this is the exception rather than the norm, implying that network coverage is generally unaffected.



(a) t=15 (b) t=30 Legend

(c) t=75 (d) t=150 (e) t=300

(f) t=450 (g) t=650 (h) t=900

Figure 26. Nodes sleeping in the network over time. Triangles are sleeping nodes, dots have at least 1 page

9.2 Pseudo-code for 3 Phases in Freshet

The pseudo-code in Figure 27 and Figure 28 below illustrates the 3 phases of operation of Freshet. Some of the aspects that Freshet borrows from Deluge (like advertisement and data message suppressions) are not mentioned in the pseudocode. In Figure 27, line 1 discusses the blitzkrieg phase. In line 1a, the node has heard the warning message, so it updates its internal code data. It records the hopCount, code version, and number of pages in the code based on the warning message. In line 1b, the node then increments the hopCount and sends the message forward. Lines 1c through 1e show how the node determines whether it will sleep. If the node is

more than 3 hops away from the origin node, then the node will sleep; otherwise the node stays awake and participates in normal code transfer.

Lines 3 and 4 illustrate a case where the blitzkrieg and distribution phases occur in succession .Lines 5 through 6 illustrate the distribution phase. Once a node is close enough to download the new code, line 3 may occur. The node hears the advertisement for the new code and then propagates that information throughout the network. The node then requests the needed code page and begins downloading.

Line 5 discusses the case where the advertisement the node hears is for code it already has, in which case the node sends an advertisement with the code it has. It then waits for nodes to request the needed code.

```
1. if (warning message heard)
      a. Upgrade version of code, update number of pages needed, record hopCount
      b. Increment hopCount and send warning message with same code information
      c. if hopCount > 3
            i. Sleep for SleepFactor*(hopCount-3)
      d. else
            i. Stay awake for normal code transfer
      e. endif
2. endif
3. if (advertisement for new code heard)
      a. Upgrade version of code, update number of pages needed
      b. Propagate warning message with code version, number of pages, origin node, hopCount 0
      c. Request needed code pages and enable normal Deluge
4. endif
5. if (updated advertisement not heard)
      a. Send advertisement message with code version, page number
      b. Wait for code request
      c. Initiate code transfer once request is received
6. endif
```

Figure 27. Pseudo-code for a node in the blitzkrieg and the distribution phases. Lines 1-2 correspond to the blitzkrieg phase, lines 3-4 correspond to the blitzkrieg phase followed immediately by the distribution phase, and lines 5-6 to the distribution phase.

Figure 28 illustrates the third phase of Freshet, the quiescent phase. The quiescent phase is initiated only after 6 or more redundant advertisements are heard, in which case the node

assumes that every node in its vicinity has the most recent download, so it is now reasonable to sleep. Lines 5a and 5b show that as the node has more neighbors it is more likely to sleep for a predefined interval $\tau/2$.

```
1.  if (heard redundant advertisements)
        a.  R++
        b.  Call TestQuiescencePhase(R)
2.  endif
3.  TestQuiescencePhase(R)
4.  {
5.  if (R > 5)
        a.  Choose random number from 0 to 1
        b.  If Rand > 1-1/b_N then Sleep for advertisement period τ/2
6.  endif
7.  }
```

Figure 28. Pseudo-code for a node in the quiescent phase. It commences after 6 redundant cycles of advertisements (no new code or nodes needing code). R is the number of redundant advertisements heard, N is the number of neighbors.