A Lightweight and Density-Aware Reprogramming Protocol for Wireless Sensor Networks

Wei Dong, *Student Member*, *IEEE*, Chun Chen, *Member*, *IEEE*, Xue Liu, *Member*, *IEEE*, Jiajun Bu, *Member*, *IEEE*, and Yi Gao, *Student Member*, *IEEE*

Abstract—We propose ReXOR, a lightweight and density-aware reprogramming protocol for wireless sensor networks using XOR. It employs XOR encoding in the retransmission phase to reduce the communication cost. In sparse and lossy networks, it delivers much better performance than Deluge, a typical reprogramming protocol for sensor networks. Compared to prior coding-based reprogramming protocols, it has two salient features. First, it is computationally much more lightweight than previous coding-based reprogramming protocols using Random Linear Codes or Fountain Codes. Second, it is density-aware by adapting its interpage waiting time. Hence, it achieves good performance in both dense and sparse networks. We have implemented ReXOR based on TinyOS and evaluate its performance extensively. Results show that ReXOR is indeed lightweight compared with previous coding-based reprogramming protocols in terms of computation overhead. The results also show that ReXOR achieves good network-level performance in both dense and sparse networks, compared with Deluge and a typical coding-based reprogramming protocol, Rateless Deluge.

Index Terms—Wireless sensor networks, network protocols.

1 Introduction

Wireless sensor networks (WSNs) have been studied for a wide range of application areas, such as military surveillance, habitat monitoring, infrastructure protection, etc. WSN applications often need to be changed after deployment for a variety of reasons, such as reconfiguring a set of parameters, modifying tasks of individual nodes, and patching security holes. Many large-scale WSNs, however, are deployed in environments where physically collecting previously deployed nodes is either very difficult or infeasible. Over-the-air reprogramming is a crucial technique to address such challenges [1].

The critical service required to enable over-the-air reprogramming is a reliable code dissemination protocol (i.e., reprogramming protocol). Deluge [2] is a highly optimized reprogramming protocol for TinyOS [3]. It uses a NACK-based protocol for reliability, and employs segmentation (into pages) and pipelining for spatial multiplexing. However, in sparse and lossy networks, the performance of Deluge degrades due to the need for a large amount of data retransmissions. To address this issue, several recent studies [4], [5], [6] employ network coding to reduce the number of transmissions, i.e., packets are encoded before they are transmitted, and incremental redundancy is used to recover from losses, reducing the retransmission cost considerably.

Previous coding-based reprogramming protocols, however, are not lightweight in terms of computation overhead [4], [5], [6]. Specifically, Rateless Deluge [4], SYNAPSE [5], and AdapCode [6] all require Gaussian elimination for decoding. Gaussian elimination is expensive in terms of computation time. For example, it was reported in [4] that the decoding delay of Rateless Deluge for TMote Sky is 1.96 seconds for a page consisting of 24 packets, and will increase to 6.96 seconds for a page consisting of 48 packets. As sensor nodes are not envisioned to be equipped with much more capable CPUs with energy, cost, and form factors being the emphasized considerations [7], a long encoding or decoding delay will very likely be a bottleneck in terms of the overall completion time.

Moreover, previous coding-based reprogramming protocols are not density-aware. In dense networks, they are not efficient because of the existence of highly variable link qualities where nearby neighbors have high link qualities and nodes further away have poor link qualities [2]. In this case, the data redundancy of previous coding-based reprogramming protocols is dominated by the poor links. It is more efficient to propagate to nearby neighbors as quickly as possible, thus further away neighbors can get better link qualities, which, in turn, reduces packet losses and retransmissions.

In order to address the deficiency of Deluge in sparse and lossy networks and the deficiency of previous coding-based reprogramming protocols, especially, in dense networks with highly variable link qualities, we propose ReXOR, a lightweight and density-aware reprogramming protocol for WSNs using XOR. It employs XOR encoding [8]

- *W. Dong, C. Chen, J. Bu, and Y. Gao are with the College of Computer Science, Yuquan Campus, Zhejiang University, Zheda Road 38, Hangzhou 310027, China. E-mail: {dongw, chenc, bjj, gaoyi}@zju.edu.cn.*
- *X. Liu is with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, 104 Schorr Center, Lincoln, NE 68588-0150. E-mail: xueliu@cse.unl.edu.*

Manuscript received 22 Aug. 2009; revised 14 Aug. 2010; accepted 19 Oct. 2010; published online 17 Dec. 2010.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2009-08-0347. Digital Object Identifier no. 10.1109/TMC.2010.240.

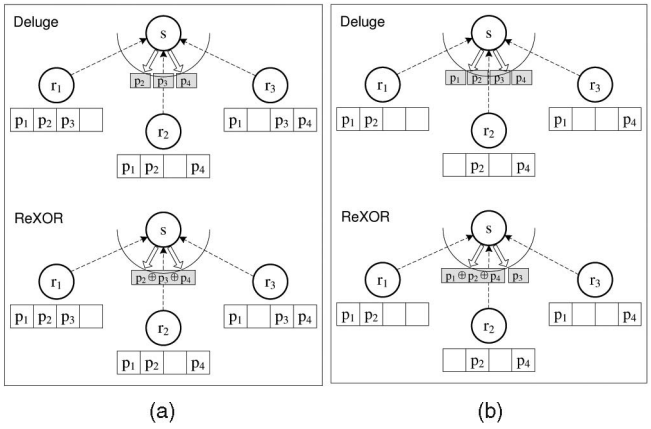

Fig. 1. Examples in a sparse and lossy network. The gray boxes represent packets retransmitted by the sender. The blank boxes indicate that the corresponding packets in a page are lost.

in the retransmission phase to enhance the performance of Deluge in sparse and lossy networks. We use XOR to transmit an encoded packet that can be decoded at multiple recipients, leveraging the fact that wireless receptions at different nodes are known to be highly independent [9], [10]. Compared to previous coding-based reprogramming protocols, it has two salient features. First, it is computationally more lightweight than previous coding-based reprogramming protocols using Random Linear Codes [4] or Fountain Codes [5]. In the initial transmission phase, ReXOR does not employ coding to avoid the decoding complexity at the receivers. In the retransmission phase, ReXOR uses coding to improve the error recovery efficiency. Specifically, ReXOR transmits XOR-ed packets that can be decoded instantly at multiple receivers, eliminating the use of Gaussian elimination which incurs a large computation overhead for resource-constrained sensor nodes. Second, it is density-aware by adapting the interpage waiting time. In dense networks with high link variance, we decrease the interpage waiting time to shorten the completion time. In sparse and lossy networks, we increase the interpage waiting time to improve coding opportunities, so as to reduce the transmission cost.

We have implemented ReXOR based on TinyOS and evaluate its performance extensively. Results show that ReXOR is indeed lightweight compared with previous coding-based reprogramming protocols in terms of computation overhead. The results also show that ReXOR achieves good network-level performance in both dense and sparse networks, compared with Deluge [2] and a typical coding-based reprogramming protocol, Rateless Deluge [4].

The rest of this paper is organized as follows: Section 2 provides illustrative examples that motivate our design. Section 3 gives an overview of ReXOR. Section 4 presents the design principles. Section 5 describes the implementation details. Section 6 shows the evaluation results. Section 7 discusses the related work. Finally, we conclude this paper in Section 8.

2 Motivating Examples

In this section, we use illustrative examples to explain the intuition underlying ReXOR's design.

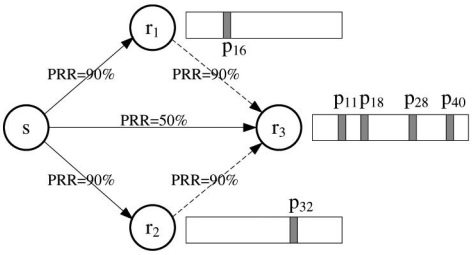

Fig. 2. An example in a dense network with high link variance. The gray bars represent the lost packets. PRR denotes the packet reception rate over a link.

2.1 Sparse and Lossy Networks

The examples in this section illustrate the advantages of ReXOR over Deluge in terms of data traffic in sparse and lossy networks. We consider such a network in which sensor nodes two hops away can be considered as disconnected. Fig. 1 shows a one-hop topology consisting of a base station and three sensor nodes. 1) In Fig. 1a, suppose that the base station broadcasts four data packets and each sensor node in the network fails to receive a different packet. In Deluge, each sensor node will report a NACK packet (i.e., reception report) for requesting the missing packets and the base station needs to retransmit the union of all lost packets, i.e., p_2, p_3, and p_4. XOR encoding can reduce the number of data transmissions. In this example, after learning each sensor node's reception status, the sender only needs to retransmit one packet, i.e., $p_1 \oplus p_2 \oplus p_3$, at once. Hence, XOR encoding reduces the number of data retransmissions from three to one. 2) As another example, in Fig. 1b, suppose that the base station broadcasts four data packets and each sensor node in the network fails to receive two packets. In Deluge, the base station needs to retransmit all four packets. On the other hand, with XOR encoding, the sender only needs to retransmit two packets, i.e., $p_1 \oplus p_2 \oplus p_4$ and p_3, after learning each sensor node's status. Hence, in this example, XOR encoding reduces the number of data retransmissions from four to two.

The key insight of ReXOR's improvement over Deluge is that it employs XOR encoding to reduce the number of retransmissions in sparse and lossy networks. It is worth noting that in a lossy network, the transmission of a page consists of multiple rounds. A sender finishes the current round of retransmission when all requested packets have been sent out. If a receiving node still loses some packets in a page, it will keep sending requests to the sender, which initiates another round of retransmission. As such, the protocols ensure 100 percent reliability.

2.2 Dense Networks with High Link Variance

The example in this section illustrates the advantages of ReXOR over a typical coding-based reprogramming protocol, Rateless Deluge [4], in terms of completion time in dense networks with high link variance. We consider such a network in which there exist nearby neighbors with good link qualities and further away neighbors with poor link qualities. Fig. 2 shows a part of such a topology consisting of a base station and three sensor nodes. Suppose that after the base station finishes one round of transmission, two

TABLE 1
Definitions Used in This Paper

Terms	Definition
Native packet	Uncoded packet
Encoded packet or XOR-ed packet	A packet that is the XOR of multiple native packets
Packet degree	The number of native packets which are XOR-ed together in a given page
NACK, reception report or request vector	A bit vector describing which packets in a given page are missing and thus requested. The missing packets are represented by 1s and packets already received are represented by 0s.
Encode vector	A vector describing which native packets are XOR-ed to form the given encoded packet

nearby neighbors, r_1 and r_2, lose a different packet while the further away neighbor, r_3, loses more packets. We consider r_3's completion time. First, we stress that in dense networks, it is very important to leverage nearby nodes as forwarders to further away nodes. For example, without leveraging nearby nodes, it would require $4/50\% = 8$ transmissions for r_3 to receive the current page. In contrast, it would require at most $1/90\% + 4/90\% \approx 5.5$ transmissions for r_3 to receive the current page, leveraging r_1 or r_2 for forwarding. Second, we consider how well we can leverage nearby nodes. We assume the decoding delay for Rateless Deluge is $d = 1.96$ seconds [4]. The expected time for transmitting a single packet considering MAC backoff is approximately $t = \frac{10}{2} + 1.3 = 6.3$ ms (where 10 ms is the maximum initial backoff time in TinyOS CSMA MAC [3] and 1.3 ms is the time for transmitting a packet for the CC2420 radio [11]).

1) In Rateless Deluge, in the first phase, s transmits to r_1 (or r_2) an encoded packet which can be overheard by r_3 with probability 0.5. This phase requires $(1/90\%)t \approx 1.1t$ ms. In the second phase, r_1 first decodes the page (which requires 1.96 seconds) and then transmits the remaining packets to r_3 (which requires $((4-1)/90\%)t \approx 3.3t$ when r_3 overhears one packet in the first phase with probability 0.5, and requires $(4/90\%)t \approx 4.4t$ when r_3 misses the packet in the first phase with probability 0.5). Hence, in average, the second phase requires $d + 3.3t \times 50\% + 4.4t \times 50\% = d + 3.85t$. Considering the two phases, Rateless Deluge costs $1.1t + d + 3.85t = d + 4.95t \approx 2.02$ seconds for r_3 to receive the page. 2) If we do not employ packet encoding for fast propagation, it would require about $(1/90\%)t + (4/90\%)t \approx 5.5t \approx 61$ ms for r_3 to receive the page, shortening r_3's completion time by $d + 4.95t - 5.5t = d - 0.55t \approx 1.96$ seconds compared to Rateless Deluge.

The key insight of ReXOR's improvement over Rateless Deluge in dense networks is that by decreasing the interpage waiting time for fast propagation, we can leverage nearby nodes as forwarders to further away nodes, reducing the number of retransmissions and shortening the completion time. Note that Rateless Deluge cannot do so because its interpage waiting time is lower bounded by the decoding delay.

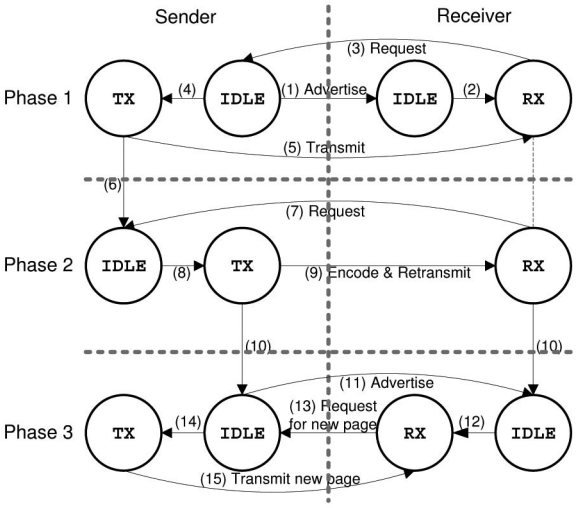

Fig. 3. ReXOR's state transition diagram.

3 ReXOR Overview

ReXOR is a code dissemination protocol for WSNs that is based on Deluge [2]. Compared to prior works, it has two important features. First, it leverages the spatial diversity by using a lightweight XOR coding scheme in the retransmission phase to reduce the communication cost. Second, it exploits the adaptive behavior by determining the interpage waiting time in a density-aware manner to achieve good performance in both dense and sparse networks.

The intuition behind ReXOR's density-aware adaptation technique (i.e., determining the interpage waiting time according to the network density) is that in sparse and lossy networks, we increase the interpage waiting time for collecting more reception reports, so as to improve the coding opportunity while in dense networks with high link variance, we decrease the interpage waiting time for fast propagation. In dense networks, with a short interpage waiting time, it is highly probable that nearby nodes will complete a page very quickly. Hence, these nodes can, in turn, serve other further away nodes, which reduces packet retransmissions and propagation time.

Like Deluge, ReXOR works in an epidemic fashion to deal with multihop transmission. All network nodes advertise about their states. When a new program image is found, nodes send requests to obtain the image. Table 1 defines the terms used in the rest of this paper. Fig. 3 depicts the state transition diagram of ReXOR in three different phases, which will be described in detail in the following three sections.

3.1 Phase 1: Advertisement, Request, and Page Transmission

Initially, each node is in the `IDLE` state, advertising about its local images periodically by a Trickle timer [12]. We use "image" to refer to the program image that needs to be disseminated and reprogrammed onto all nodes in the network. When a node (receiver) learns that another node (sender) has an image that has more number of available pages, it will send a request to the sender. It then enters into the `RX` state, preparing to receive the data packets. When the sender receives a request, it will transit

from the IDLE state to the TX state. Then, it starts transmitting the requested data packets in the current page. For example, we assume that an image has seven pages. During the dissemination, the image at different nodes may not be complete. Assume that a sender has six available pages, and a receiver has five available pages. In the advertising process, the receiver will find that the sender has more available pages. Hence, it requests to the sender for the last page. After receiving the request, the sender will start the transmission process. Also note that both Deluge and ReXOR enforce strict ordering of page transmission, e.g., if the receiver has the first five (i.e., from pages 1 to 5) pages available, it must receive page 6 before it can receive page 7.

3.2 Phase 2: Page Retransmission

The sender will enter into the IDLE state whenever it finishes transmitting all requested packets in the current page. If a receiver loses some packets in the page, it will remain in the RX state, sending requests to one of the senders for the missing packets. When a sender receives a request for retransmission purpose, it delays an adaptive interpage waiting time, trying to collect multiple reception reports. Indeed, the value of the waiting time is a trade-off between data traffic and propagation delay. We dynamically adapt the interpage waiting time in a density-aware manner to achieve good performance in both sparse and dense networks. We will discuss the details in Section 4.3.

After the sender collects a number of reception reports, it decides which packets should be XOR-ed to minimize the number of retransmissions. To achieve this goal, the sender executes a fast coloring algorithm. After the algorithm is executed, packets with the same color are XOR-ed and retransmitted. For decoding purpose, the sender attaches a variable header containing the encode vector. After the transmission of all requested packets, the sender enters into the IDLE state. We will elaborate on the encoding details in Section 4.1.

After receiving an encoded packet, the receiver decodes it by XOR-ing packets which are already received. There are chances that a receiver cannot decode a packet because its reception report is not heard by the sender. In this case, the receiver simply discards the packet. Our coloring algorithm ensures that the XOR-ed packets can be instantly decoded (without Gaussian elimination) by all receivers whose reception reports are heard by the sender. We will discuss the decoding process in Section 4.2.

When the receiver still loses some packets in the current page, it remains in the RX state, sending requests to one of the senders, which initiates another round of retransmission. Finally, when all the missing packets in the current page are received, the receiver enters into the IDLE state and prepares to receive the next page.

3.3 Phase 3: New Page Transmission

When the receiver finishes receiving the current page, it increases its advertising frequency by resetting its advertisement timer.

When further away nodes learn that the receiver has more available pages, the receiver can serve as a sender itself. This allows the last received page to be propagated in a multihop manner. We keep the pipelining feature of Deluge: whenever a node finishes receiving a page, it can serve further away nodes which have fewer available pages.

Both Deluge and ReXOR enforce the transmission of a page with a lower page number with higher priority in a neighborhood: when any node in a neighborhood learns that a page with lower page number is requested or is currently transmitted, it will postpone a request for transmission of a page with a higher page number.

When the current page is received by all nodes in a neighborhood, the receiver will request to the sender for a new page, after learning that the sender has more available pages. After the sender receives a new page request, it does not wait for other reception reports. This is because additional reception reports from other nodes will not reduce the number of transmissions in this phase. Therefore, the sender goes to the TX state and trying to transmit the new page when the channel is free.

4 ReXOR Design

In order to make ReXOR work, we need to address some important system design issues. We will discuss each of them in the following sections.

4.1 Packet Encoding

Packet encoding is employed in ReXOR in the page retransmission phase. To achieve this goal, the sender delays an adaptive interpage waiting time for collecting multiple reception reports. Before we introduce the encoding details, let us first define the following notations.

- p_i is the ith packet in a given page. A page consists of M packets. i denotes the packet index within the given page ($1 \leq i \leq M$).
- P is a set of packets in a given page, i.e., $P = \{p_{i_k}\}_{k=1}^{m}$, where i_k is the packet index.
- r is a request vector. It consists of M bits where 1 denotes the corresponding packet is lost and thus requested; 0 denotes the corresponding packet is received. For example, if a page consists of four packets, i.e., $M = 4$, a request vector $r = (1100)$ represents that the first two packets (i.e., p_1 and p_2) are lost and hence requested while the last two packets (i.e., p_3 and p_4) are correctly received. We use $r[i]$ ($1 \leq i \leq M$) to denote the ith bit in request vector r. In the example above, $r[1] = r[2] = 1$ and $r[3] = r[4] = 0$.
- Each request vector r is sent by a requester to the sender. We order the request vectors received by the sender according the time of reception. Hence, r_i is the ith request vector received by the sender ($1 \leq i \leq n$, where n is the total number of distinct request vectors received by the sender). Note that in the ordered list $(r_1, \ldots, r_n)$, the request vectors sent by the same requester appear only once. For example, the sender has already received r_1 and r_2. Then, it receives another request vector. If this request vector and r_1 are both sent by the same requester, we just ignore the currently received request vector, i.e., we do not append the current request vector to the ordered list, so the ordered list is still (r_1, r_2).

- R is a set of request vectors collected by the sender, i.e., $R = \{r_i\}_{i=1}^{n}$. $R[i]$ is the bitwise OR value of the ith bits in all the request vectors contained in R, i.e., $R[i] = \bigvee_{r \in R} r[i]$. $R[i] = 0$ implies that all elements in R have p_i available while $R[i] = 1$ implies that there exists at least one element that has lost p_i and thus requests for p_i.
- R^P is a set of request vectors. Each element in R^P is a request vector which requests for at least one packet in P. For simplicity of notation, R^i denotes the set of request vectors within which each element requests for packet p_i. By definition, $R^P = \bigcup_{k=1}^{m} R^{i_k}$, where $P = \{p_{i_k}\}_{k=1}^{m}$.

After receiving a set of request vectors $(r_1, \ldots, r_n)$, the sender must decide on which packets should be XOR-ed to minimize the number of retransmissions. To permit instant decoding at all requesters, the sender can XOR m (≥ 2) native packets together only if each requester has all $m-1$ native packets available other than the native packet it requests for (so that it can retrieve the native packet it requests for). This problem is formally expressed as follows:

Definition 1 (Minimum Packet Transmission Problem). *Given n requests vectors, $r_1, \ldots, r_n$, the minimum packet transmission problem is to find the minimum number of (encoded) packets for transmission, in order to satisfy all n requests such that each requester can instantly decode the received packets. A packet can be instantly decoded at a receiver if it is an XOR of native packets containing at most one native packet that the receiver has not decoded yet.*

As an example, consider the case shown in Fig. 1b. Suppose the sender receives three request vectors successfully, i.e., $r_1 = (0011)$, $r_2 = (1010)$, and $r_3 = (0110)$. The simplest scheme is to retransmit all four packets. One could also retransmit three packets, say, $p_1 \oplus p_2$, p_3, and p_4. The most efficient scheme is to retransmit only two packets, i.e., $p_1 \oplus p_2 \oplus p_4$ and p_3. Note that transmitting only one encoded packet, i.e., $p_1 \oplus p_2 \oplus p_3 \oplus p_4$ ($m = 4$), does not meet the requirement of the problem. This is because the encoded packet cannot be instantly decoded at any of the requesters as none of the requesters has $m - 1 = 3$ native packets available. To address the above Minimum Packet Transmission Problem (MPTP), we need to find the most efficient encoding scheme that minimizes the number of transmissions while allowing instant decoding at all requesters.

This problem, similar to the MIN-T-2 problem as described in [13], is $\mathcal{NP}$-complete. We have also proved that this problem, equivalent to the Minimum Clique Partition Problem (MCPP)[14], is $\mathcal{NP}$-complete. Moreover, it is inapproximatable within $n^{1-\varepsilon}$ for any $\varepsilon > 0$, unless $\mathcal{ZPP} = \mathcal{NP}$. We will show the proofs in the following text.

As mentioned above, to permit instant decoding at a requester, the sender can XOR m native packets together only if each requester has all $m-1$ native packets other than the native packet it requests for. This observation can be expressed by the following proposition:

Proposition 1. *We consider m native packets within a given page. The indexes of these packets are denoted as $i_1, \ldots, i_m$, respectively. P denotes the set of the m native packets, $P = \{p_{i_k}\}_{k=1}^{m}$. These m native packets can be XOR-ed by the sender, denoted as $X(P)$, if and only if, $\forall r \in R^P$, r requests and only requests one native packet in P.*

As implied by the above proposition, two native packets, p_i and p_j, can be XOR-ed if and only if all request vectors that request for p_i have p_j available (i.e., $R^i[j] = 0$), and all request vectors that request for p_j have p_i available (i.e., $R^j[i] = 0$). For example, consider the second example in Section 2.1. First, we consider whether p_1 and p_2 are XOR-able. We notice that request vectors request for p_1, i.e., r_2, have p_2 available. So r_2 can decode $p_1 \oplus p_2$ for retrieving p_1. On the other hand, request vectors request for p_2, i.e., r_3, have p_1 available. So r_3 can decode $p_1 \oplus p_2$ for retrieving p_2. Hence, p_1 and p_2 are XOR-able (it can be decoded at all three receivers). Second, we consider whether p_1 and p_3 are XOR-able. We notice that request vectors request for p_1, i.e., r_2, have p_3 lost. So if p_1 and p_3 are XOR-ed, the encoded packet cannot be decoded at r_2. Hence, p_1 and p_3 are not XOR-able. Based on the above observation, we get the following lemma:

Lemma 1. $X(\{p_i, p_j\}) \Leftrightarrow R^i[j] = R^j[i] = 0$.

Following the above lemma, we continue to prove the necessary and sufficient condition under which a set of packets can be XOR-ed.

Theorem 1. *$X(P)$, where $P = \{p_{i_k}\}_{k=1}^{m}$ iff $\forall l, j \in \{i_k\}_{k=1}^{m}$ $(l \neq j)$, $X(\{p_l, p_j\})$.*

Proof. We will prove this theorem by examining both the necessary condition and the sufficient condition.

Necessary condition. By definition, $R^P = \bigcup_{k=1}^{m} R^{i_k}$, we consider each R^l where $l \in \{i_k\}_{k=1}^{m}$. By definition, the requesters in R^l already request for p_l. By Proposition 1, they does not request for any other packets in P because packets in P can be XOR-ed (i.e., $X(P)$). In other words, $R^l[j] = 0, \forall j \neq l$. By Lemma 1, p_l and p_j can be XOR-ed, i.e., $X(\{p_l, p_j\})$.

Sufficient condition. By Lemma 1, $X(\{p_l, p_j\})$ implies that $R^l[j] = R^j[l] = 0$. Hence, we have $R^l[j] = 0$, $\forall l, j \in \{i_k\}_{k=1}^{m} (l \neq j)$. We consider whether packets in P can be XOR-ed into one encoded packet. By definition, $R^P = \bigcup_{k=1}^{m} R^{i_k}$. We check each R^l, $l \in \{i_k\}_{k=1}^{m}$. As we already have $R^l[j] = 0$, $\forall j \neq l$. This means that $\forall r \in R^l$, r only requests one packet (i.e., p_l) in P. Since l can be any value in $\{i_k\}_{k=1}^{m}$, $\forall r \in R^P = \bigcup_{k=1}^{m} R^{i_k}$, r requests only one packet in P. By Proposition 1, packets in P can be XOR-ed, i.e., $X(P)$. □

To see the correctness of the proof, we use examples to check both the necessary condition and sufficient condition.

1. Necessary condition. Consider the example in Fig. 1a. In this case, $P = \{p_2, p_3, p_4\}$, $R = R^2 \cup R^3 \cup R^4$, e.g., R^2 denotes the request vectors that request for packet p_2 (R^2 contains one element, i.e., the request vector sent by r_3). As we know that R^i ($2 \leq i \leq 4$) request for p_i (by definition), and packets in P can be XOR-ed (by the condition), $R^i[j] = 0$ ($2 \leq i \leq 4$, $1 \leq j \leq 4$, $j \neq i$) (by Proposition 1). For example, $R^2[1] = R^2[3] = R^2[4] = 0$ (the requester vector that requests for p_2 has the remaining three packets available). Similarly, $R^3[1] = R^3[2] = R^3[4] = 0$, $R^4[1] = R^4[2] = R^4[3] = 0$.

Hence, by Lemma 1, for any two packets in P, they can be XOR-ed, i.e., $X(\{p_2, p_3\})$, $X(\{p_2, p_4\})$, $X(\{p_3, p_4\})$.

2. Sufficient condition. We consider the example in Fig. 1a again. We know that $X(p_l, p_j)$ $(2 \leq l, j \leq 4, l \neq j)$, i.e., $X(p_2, p_3)$, $X(p_2, p_4)$, $X(p_3, p_4)$. By Lemma 1, $R^2[3] = R^3[2] = R^2[4] = R^4[2] = R^2[3] = R^3[2] = 0$. In order to apply Proposition 1, we check each request vector $r \in R = R^2 \cup R^3 \cup R^4$. Without loss of generality, consider $r \in R^2$, because $R^2[3] = R^2[4] = 0$, r can decode the packet $p_2 \oplus p_3 \oplus p_4$ (for retrieving p_2). Similarly, $\forall r : r \in R^3$ or $r \in R^4$, r can decode $p_2 \oplus p_3 \oplus p_4$. To summarize, $\forall r \in R^2 \cup R^3 \cup R^4 = R$, r can decode $p_2 \oplus p_3 \oplus p_4$. Hence, p_2, p_3, p_4 can be XOR-ed, i.e., $X(P)$ $(P = \{p_2, p_3, p_4\})$.

Theorem 2. *MPTP is $\mathcal{NP}$-complete, and it is inapproximable within $n^{1-\varepsilon}$ for any $\varepsilon > 0$, unless $\mathcal{ZPP} = \mathcal{NP}$.*

Proof. Consider a graph $G = (V, E)$ where the vertices represent the requested packets that need to be transmitted, and an edge between two vertices indicates that the corresponding packets can be XOR-ed.

As implied by Theorem 1, a clique (i.e., a complete subgraph) in the graph indicates that the corresponding packets can be encoded in a single packet. A clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge. Cliques are one of the basic concepts of graph theory and are used in many other mathematical problems and constructions on graphs. The MPTP problem is to find the encoding scheme that results in the minimum number of transmissions. It is equivalent to the classical MCPP problem. In the MCPP problem, we partition V into disjoint subsets $V_1, V_2, \ldots, V_k$ such that each V_i $(1 \leq i \leq k)$ is a complete graph (V_i corresponds to packets that can be XOR-ed). We need to find the partition such that the number of disjoint subsets V_i is minimized.

It is shown that the MCPP problem is again equivalent to the minimum graph coloring problem [14]. It is proved that the minimum graph coloring problem is $\mathcal{NP}$-complete, and in general graph, it is inapproximable within $n^{1-\varepsilon}$ for any $\varepsilon > 0$, unless $\mathcal{ZPP} = \mathcal{NP}$ [14]. Therefore, the theorem holds. □

Consider the example shown in Fig. 1a. Here, $V = \{p_2, p_3, p_4\}$. Next, we consider whether there should be an edge between, say, p_2 and p_3. We observe that $R^2[3] = R^3[2] = 0$, hence, p_2 and p_3 can be XOR-ed (by Lemma 1). Therefore, there is an edge between p_2 and p_3. Similarly, we check the remaining edges in the graph, and Fig. 4 (left) shows the final result.

Consider another example shown in Fig. 1b. Here, $V = \{p_1, p_2, p_3, p_4\}$. We observe that $R^1[2] = R^2[1] = 0$, so there exists an edge between p_1 and p_2. The request vector sent by r_1 is contained in R^3 (because it requests for p_3), and it requests for p_4, hence $R^3[4] = 1$ (as $R^3[4]$ is the bitwise OR value of the 4th bits in all request vectors in R^3). Hence, there does not exist an edge between p_3 and p_4. Similarly, we check the remaining edges in the graph, and Fig. 4 (right) shows the final result.

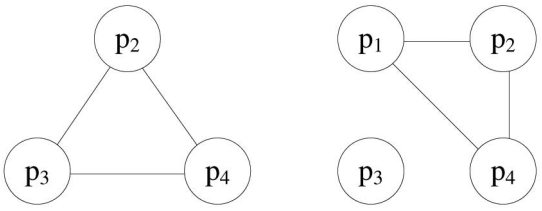

Fig. 4. The constructed graphs corresponding to the examples shown in Fig. 1.

The MCPP problem is to find the clique partition that generates the minimum number of disjoint subsets. For example, in Fig. 4 (right), a possible clique partition of V is $V_1 = \{p_1, p_2\}$, $V_2 = \{p_3\}$, $V_3 = \{p_4\}$. This generates a total of three disjoint subsets (correspond to three transmitted packets). Another clique partition of V is $V_1 = \{p_1, p_2, p_4\}$, $V_2 = \{p_3\}$. It is easy to see that this is the clique partition that generates the minimum number of disjoint subsets.

There are a number of heuristic algorithms proposed in the literature [15], [16], [17], [18], [19]. Among these, we employ an efficient sequential coloring algorithm that does not ensure a result with the minimum number of colors/transmissions (one color represents one encoded packet). It is appropriate in our case due to two reasons. First, any complex algorithm will incur a large computation overhead which is either infeasible on current sensor nodes or is not desired for quick propagation. Second, the optimal solution is not so important as we execute the algorithm online with high network dynamics.

Algorithm 1 presents the coloring algorithm. The input is a set of request vectors from neighboring nodes and the output is the colored (native) packets that need to be retransmitted. The algorithm works as follows: We iterate for all the requested packets and associate each packet with a color so that packets with the same color can be XOR-ed. Because there are $|P|$ packets in total, there are at most $|P|$ number of different colors. When we iterate for each requested packet, we test whether a particular color can be associated with that packet. The procedure C-COLOR-P? is for this purpose. If a particular color can be associated with that packet, we add the packet to the colored packet set Y_c in which all packets are associated with color c. Next, we look at the procedure C-COLOR-P?. This procedure tests whether color c can be associated with packet p_i. According to Theorem 1, and proof in Theorem 1, p_i and each packet in Y_c can be XOR-ed into one packet if and only if any two packets in $Y_c \cup \{p_i\}$ can be XOR-ed (i.e., $Y_c \cup p_i$ is a complete graph). Since we keep that any two packets in Y_c can be XOR-ed during program execution (i.e., Y_c is a complete graph), in the current iteration, we only need to check whether p_i can be XOR-ed with any packet in Y_c. The procedure P-XOR-Q? is used to test whether any two packets p_i and p_j can be XOR-ed. According to Proposition 1, two packets can be XOR-ed if and only if all requesters request at most one packet out of the two packets. That means that the two packets can be XOR-ed if and only if there does not exist a requester that requests both packets.

Algorithm 1. The Sequential Coloring Algorithm

Input: A set of request vectors R; A set of requested pkts P

Output: $Y = \bigcup_c Y_c$ where Y_c is the set of pkts with color c

1: **procedure** COLORING

```
2:    for p_i ∈ P do
3:        for c ← 1, |P| do
4:            if C-COLOR-P?(c, p_i, Y_c) then
5:                Y_c ← Y_c ∪ {p_i}        ▷ Color packet p_i with c
6:                break

7:  procedure C-COLOR-P?(c, p_i, Y_c)   ▷ Test if c can color p_i
8:    for q_j ∈ Y_c do
9:        if not P-XOR-Q?(p_i, q_j) then
10:           return FALSE
11:   return TRUE
12: procedure P-XOR-Q?(p_i, q_j)      ▷ Test if p_i and q_j can be
    XOR-ed
13:   for r ∈ R do
14:       if r[i] = 1 and r[j] = 1 then
15:           return FALSE
16: return TRUE
```

As an example, we consider the case shown in Figs. 1b and 4 (right). We first consider p_1 and associate it with color 1. After the first iteration, $Y_1 = \{p_1\}$. Next, we consider p_2 and try to associate it with color 1. We find that p_2 can be XOR-ed with all elements in Y_1, i.e., p_1, hence put p_2 in Y_1. After the second iteration, $Y_1 = \{p_1, p_2\}$. After that, we consider p_3 and try to associate it with color 1. We find that p_3 cannot be XOR-ed with $p_1 \in Y_c$, thus p_3 cannot be put in Y_1. We associate p_3 with color 2, and put it in Y_2. After the third iteration, $Y_1 = \{p_1, p_2\}$ and $Y_2 = \{p_3\}$. Finally, we consider p_4 and try to associate it with color 1. We find that p_4 can be XOR-ed with all elements in Y_1, i.e., p_1 and p_2, hence put p_4 in Y_1. Therefore, the final result is $Y_1 = \{p_1, p_2, p_4\}$ and $Y_2 = \{p_3\}$.

Our coloring algorithm ensures that the encoded packets transmitted by a sender can always be decoded at all requesters whose reception reports are heard by the sender. During one round of retransmission, it is possible for a sender to transmit multiple (encoded) packets. These packets have no interdependencies, i.e., subsequent packets do not depend on the previously transmitted packets for decoding. The encoded packets only require the native packets which are known to be available on the requesters (via the reception reports). This simplifies our design and eliminates the use of Gaussian elimination.

The complexity of Algorithm 1 is $O(nm^3)$, where n is the number of request vectors and m is the number of requested packets to retransmit. For reasonable sized n and m, this overhead is small. The execution of this algorithm is much faster than the decoding process in Rateless Deluge because of several reasons. First, in ReXOR, the packet payload is not involved in the computation while it is involved in the computation in Rateless Deluge. Second, ReXOR does not require the use of random number generator, which causes extra computation overhead. Third, the complexity of ReXOR is proportional to m^3 where m is the number of missing packets while in Rateless Deluge, m is the total number of packets in a page. For example, on TelosB nodes, for an average number of four reception reports (i.e., $n = 4$) and 20 percent packets loss rate (i.e., $m = 20 \times 20\% = 4$), the time needed for coloring is about 32 ms, more than 60 times faster than Rateless Deluge.

After executing Algorithm 1, we obtain a set of colored packets. Next, the encoding process is direct: packets with the same color are XOR-ed together to form an encoded packet. XOR encoding and decoding is very fast, e.g., XOR-ing two packets with 23-byte payload only takes about 0.08 ms on TelosB nodes.

There are two implementation details that worth noting here. First, we must add an encoding vector to the packet header to let the receiving node be able to decode (see Section 5.1). Second, we keep the working page in RAM for efficient encoding and decoding. Indeed, it incurs additional memory overhead. But it reduces the number of flash I/Os which are energy-hungry and incur delays. We will discuss these issues in detail in Section 5.

4.2 Packet Decoding

Packet decoding in ReXOR is very efficient. The receiver maintains the working page in RAM for decoding purpose. We let the RAM buffer holding the working page during `TX` for encoding purpose be reused during `RX` for decoding purpose. When the receiver receives an encoded packet consisting of k native packets, it goes through the indexes of the native packets one by one, and retrieves the corresponding packet from its buffer if possible. Ultimately, it XORs the $k-1$ packets with the received encoded packet to retrieve the native packet meant for it. An important feature of ReXOR is that the receiver can instantly decode a packet when it has arrived, unlike previous coding-based reprogramming protocols that require Gaussian elimination after an expected number of packets have arrived.

If a receiver still loses some packets after one round of retransmission, it will remain in the `RX` state, sending requests (i.e., NACKs) to one of the senders, which initiates another round of retransmission. The NACK mechanism ensures 100 percent reliability for code dissemination.

There are chances that a receiver cannot decode a packet if its request report is not received by the sender. In this case, we simply discard the packet, rather than buffering it in RAM for later processing. This simple design reduces RAM consumption and also eliminates Gaussian elimination, which is important for resource-constrained sensor nodes.

4.3 Adaptive Interpage Waiting

An important system design issue is how to determine the interpage waiting time for collecting multiple reception reports (or request vectors). We define the interpage waiting time (w) to be the time interval between the reception of the first reception report and the start of the next transmission (i.e., the sender transits from `IDLE` to `TX`). The choice of w is a trade-off between data traffic and propagation delay. A large value of w will increase the propagation delay but will also help to collect more reception reports for reducing data traffic. A small value of w will shorten the propagation delay but fewer packets can be encoded and subsequently corrected at the receivers in a single transmission round.

In Deluge, as soon as the sender receives the request, it enters into the `TX` state immediately (i.e., $w = 0$). In ReXOR, we determine w by considering the following factors.

First, the sender keeps on waiting until the requested page is loaded into RAM.

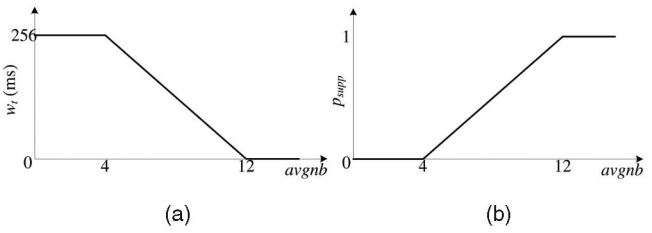

Fig. 5. (a) Relationship between w_t and $avgnb$, and (b) Relationship between p_{supp} and $avgnb$.

Second, the sender stops waiting when a request vector with missing packet number exceeds a threshold. We select this threshold as $0.8M$ ($= 16$ if $M = 20$) in our current implementation. That is, if a sender receives a request with $\geq 0.8M$ missing packets, ReXOR will start the transmission immediately (without waiting for more requests). The reason is as follows: In this case, ReXOR needs to transmit at least $0.8M$ packets (no matter how long to wait for more requests). In other words, the reduction of transmissions is at most $0.2M$ packets ($= 4$ if $M = 20$). Hence, the benefit of waiting is small. In particular, when the sender receives a new page request $r = (11, \ldots, 11)$, the sender starts the transmission immediately. This strategy makes sense because ReXOR delivers no coding gains in this case.

Third, the sender stops waiting when a threshold of requests have been received. We select this threshold as six in our current implementation. The reason is that it would be less beneficial to collect more requests when the time complexity of our coloring algorithm increases.

Finally, the sender stops waiting when the interpage waiting timer expires. We set the timer interval w_{t} adaptively according to the density. The intuition of the adaptive design is that in dense networks, it makes sense for fast propagation so that neighbors can collaborate with each other while in sparse networks, it is beneficial to collect multiple reception reports to improve the coding opportunity. For this purpose, we let each node estimate its neighbor number. Each node keeps a counter, nb, which is the number of different nodes it knows of (via messages it hears) during a fixed time window which is set to one second (i.e., twice the minimum advertisement interval). The long-term number of neighbors, $avgnb$, is calculated as follows: $avgnb = \alpha \times avgnb + (1 - \alpha) \times nb$. We currently set $\alpha = 2/3$ in our implementation. As suggested by [6], we consider $avgnb = 4$ or less as a low density and $avgnb = 12$ or more as a high density. Next, w_t is determined according to Fig. 5a. The maximum value of w_t is set to 256 ms because 256 ms is the maximum backoff time before a receiver sends requests. This means that the sender waits at most 256 ms before it starts the next transmission. The minimum of w_t is set to 0, which means that the sender stops waiting immediately when the first request is received (the same to Deluge).

4.4 Request Suppression

In order to collect request vectors from multiple neighbors, we need to slightly modify Deluge's request suppression mechanism.

In Deluge, when a node overhears a request with the same (or smaller) image number and the same (or smaller) page number, it suppresses its own request.

This mechanism, however, prevents the sender to collect multiple request vectors for a given page, which, in turn, reduces coding opportunities. Hence, in ReXOR, we slightly modify the suppression mechanism. For a given page, we do not suppress subsequent requests for the same page, instead, a node suppresses its own request if and only if it overhears a request vector (r_{overhear}) that is an superset of its own request vector (i.e., $r_{\mathrm{overhear}} \supseteq r$). The intuition behind ReXOR's superset suppression mechanism is as follows: If all the requested packets in the current page have already been requested by some other nodes (i.e., $r \subseteq r_{\mathrm{overhear}}$), it is beneficial to suppress the request because the retransmitted packets can recover the losses at r. On the other hand, if there exists nonoverlapping missing packets, the request should not be suppressed in order to let the sender learn this fact for efficient XOR encoding.

The superset suppression mechanism increases the number of request messages compared to Deluge, which may cause packet collisions, especially, in dense networks. In order to mitigate this effect, we adopt a probabilistic-based superset suppression mechanism. The basic idea is to suppress more request messages in dense networks. The probability p_{supp} for request suppression depends on the network density. In our current implementation, we set p_{supp} as shown in Fig. 5b, i.e., in dense networks with more than 12 neighbors we suppress request messages as in Deluge, while in sparse networks with less than four neighbors we suppress the request message r only when the overheard request is a superset of r.

5 Implementation

We have implemented ReXOR based on TinyOS. In this section, we describe the data packet format and the control flow of ReXOR. Finally, we summarize our implementation overhead.

5.1 Data Packet Format

In ReXOR, each data packet, regardless of whether it contains native packet or an XOR of native packets, must include an encode vector that describes the native packets from which it is formed. For example, a degree i packet must therefore contain i packet indexes for the receiver to decode the packet.

The encode vector is represented by either the $\log M$ format (similar to the $\log N$ format in [20]) or the bit format, depending on which format uses fewer bytes. We denote b as the bytes needed to encode the packet indexes. Thus, $b = \min(b_{\log M}, b_{\mathrm{bit}})$, where $b_{\log M}$ is the number of bytes for the $\log M$ format and b_{bit} is the number of bytes for the bit format.

First, we consider $b_{\log M}$. In the $\log M$ format, only the encoded packet indexes are represented (each index is encoded by a fixed number of bits). Hence,

$$b_{\log M} = \left\lceil \frac{\mathrm{degree} \times \lceil \log M \rceil}{8} \right\rceil. \quad (1)$$

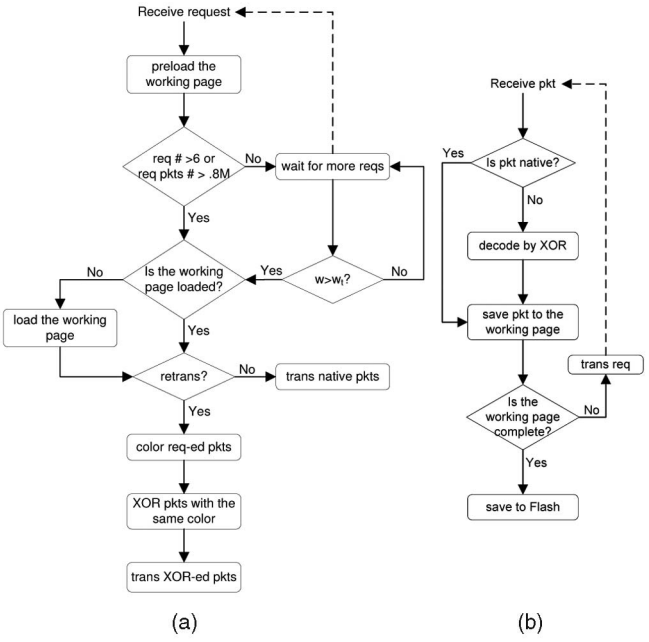

Fig. 6. Control flow of ReXOR's implementation. The dashed arrow indicates that the originated action gives rise to the pointed event. (a) sender side. (b) receiver side.

Second, we consider b_{bit}. In the bit format, 1s in the M-sized bit vector represent that the corresponding packets are encoded. Hence,

$$b_{\text{bit}} = \left\lceil \frac{M}{8} \right\rceil. \tag{2}$$

When $b_{\log M} < b_{\text{bit}}$, i.e.,

$$\text{degree} \leq \frac{(\lceil M/8 \rceil - 1) \times 8}{\lceil \log M \rceil}, \tag{3}$$

we choose the $\log M$ format while vice versa. For example, when $M = 20$, we use the $\log M$ format when $\text{degree} \leq 3$, and the bit format when $\text{degree} > 3$; when $M = 48$, we use the $\log M$ format when $\text{degree} \leq 6$, and the bit format when $\text{degree} > 6$.

5.2 Control Flow

Fig. 6 abstracts the working diagram of ReXOR. The control flow responds to request reception and data packet reception.

On the sending side (Fig. 6a), when the sender receives a request, it first tries to preload the requested page from flash to RAM. Then, it determines whether the number of requests or the number of missing packets exceed the thresholds. If the above conditions are satisfied, the sender finishes waiting immediately. Otherwise, it waits for additional requests until an adaptive time-out (w_t) has elapsed. Ultimately, the sender finishes waiting and checks again whether the working page is in RAM. After the working page is loaded, the sender checks whether it is in the retransmission phase (i.e., error recovery phase). If it is not, the sender start transmitting native packets. Otherwise, the sender first executes our coloring algorithm, and then it encodes and starts transmitting XOR-ed packets.

TABLE 2
Computation Time (ms) of the Coloring Algorithm (Algorithm 1) under Different Workloads

Loss rate\$|R|$	2	3	4	5	6
10%	4.17	5.71	14.17	17.14	22.27
20%	11.80	28.16	32.41	36.66	47.73
30%	26.62	40.58	46.16	59.22	65.06

$|R|$ *denotes the number of received requests.*

On the receiving side (Fig. 6b), when a node receives a data packet, it checks whether it is a native packet. If it is, the receiver saves the packet to the working page in RAM. Otherwise, it decodes and then saves it to the working page. Next, the receiver checks if the working page is complete. If it is not, the receiver starts transmitting requests to the most recently heard neighbor. Otherwise, it saves the working page to flash, and starts requesting for the new page.

5.3 Overhead

Finally, we estimate ReXOR's overhead and its suitability for deployment in WSNs.

5.3.1 Coding Overhead

In this section, we compare the computational performance of ReXOR with Rateless Deluge. We do not show results for Deluge as it does not employ network coding. The major contributor of computation overhead in Rateless Deluge is decoding. It was reported in [4] that the decoding process requires 1.96 seconds for a page consisting of 24 packets and 6.96 seconds for a page consisting of 48 packets. In ReXOR, the computation overheads include 1) the coloring algorithm (Algorithm 1) at the sender side, 2) the encoding time at the sender side, and 3) the decoding time at the receiver side. XOR encoding and decoding are very efficient: XORing two data packets of 23-byte payload only incurs an overhead of about 0.08 ms on TelosB nodes. The major contributor of computation overhead is incurred by the coloring algorithm.

In order to quantify the coding overhead, we write ReXOR's coloring algorithm in one separate benchmark, with varying number of request vectors (i.e., n) and varying number of missing packets (i.e., m). In order to measure the timing, we use TinyOS's 32 kHz timer. Table 2 provides the average time needed to execute the coloring algorithm under different workloads. We can see that ReXOR is computationally more lightweight than Rateless Deluge.

5.3.2 Memory Overhead

In this section, we compare the memory overhead of Deluge, Rateless Deluge, and ReXOR. For this purpose, we use the `DelugeBasic` application in the TinyOS distribution. This application has only one configuration that wires to the reprogramming protocol, and introduces no additional overhead.

Table 3 shows the ROM and RAM consumption of Deluge, Rateless Deluge, and ReXOR. The page size is set to 20 pkts/page. We can see that ReXOR's memory overhead is much smaller than Rateless Deluge. This is because Rateless Deluge incurs extra overheads such as the table of multiplicative inverses, the precoding buffer, etc. [4].

TABLE 3
Memory Overhead of Deluge, Rateless Deluge, and ReXOR (in Bytes)

	Deluge	Rateless Deluge	ReXOR
ROM	19938	27210	21354
RAM	623	3148	1633

We have also examined ReXOR's memory overhead when the page size increases to 48 pkts/page. Indeed, a large page size may be beneficial in certain network topologies [21]. We observe that the ROM size keeps almost the same (i.e., 21,376 bytes) while the RAM size increases to 2,979 bytes. This is still smaller than Rateless Deluge (with 20 pkts/page) but is much larger than Deluge because we keep the current page in RAM to speed up the encoding and decoding process. An additional benefit is that it can reduce the number of flash I/Os. For example, in a lossless scenario, for receiving an image consisting of seven pages, Deluge requires $7 \times 20 = 140$ flash writes while ReXOR only requires seven flash writes. Fewer number of flash I/Os is preferred considering the limited number of I/Os for flash.

5.3.3 Header Overhead

We need to modify the Deluge data packet format to incorporate the encoding vector. We replace the `pktNum` field of `DelugeDataMsg` by a variable byte vector. The maximum size of the byte vector is 3 for our current implementation because we choose the page size $M = 20$ and thus the bytes of the encode vector $b = \min(b_{\log M}, b_{\text{bit}}) \leq 3$. Correspondingly, the maximum packet payload length is increased from 29 to 31 bytes. It is acceptable for current sensor nodes.

6 Evaluation

In this section, we conduct testbed experiments to evaluate ReXOR's network-level performance.

6.1 Methodology

We implement ReXOR on the TinyOS/TelosB platform. We conduct experiments based on a clique topology consisting of five TelosB nodes and a grid topology consisting of 16 TelosB nodes. The clique and the grid topology are representative for two typical reprogramming scenarios: 1) Reprogramming during lab testing. In this scenario, nodes are placed near each other and are reprogrammed by a base node in a single hop. 2) Reprogramming after deployment. We think that the grid topology is a good representation of practical multihop networks.

We compare Deluge, Rateless Deluge, and ReXOR in different network topologies under different loss models. All experiments use a default page size of 20 pkts/page. We use the `CntToLeds` benchmarks for dissemination. This benchmark has seven pages in total.

We use two loss models. The first is forced packet loss, which is dropping packets uniformly at random at the receiver side. The second is natural packet loss because of the constraints of transmission power level and internode distance.

We first evaluate ReXOR's performance in the single-hop clique topology (with five TelosB nodes) with varying forced loss rates. In this case, sensors are transmitting at the maximum power level.

We then evaluate ReXOR's performance in grid topologies (with 16 TelosB nodes). In the sparse grid topology, we set the transmission power level to 2. The internode distance is approximately 25 cm. We have experimentally verified that at such a distance, one-hop neighbors can communicate perfectly while two-hop neighbors have very poor link quality. We introduce a forced loss rate of 20 percent. In the dense grid topology, we also set the transmission power level to 2, and the internode distance is approximately 12 cm. In this case, a node has much more number of neighboring nodes. There is no forced loss rate. In order to collect multiple statistics, we write additional software modules. The `Bcast` component synchronize all nodes to the base station node at the maximum transmission power. It also lets each node report its statistics (e.g., completion time and transmitted packets) at the maximum power after the dissemination process is finished. We use a sniffer node to overhear all statistics. The sniffer program is a modified version of the `PacketSniffer_802_15_4` program in the TinyOS-contrib repository.

We compare three protocols in terms of data traffic and completion time. State-of-the-art reprogramming protocol for WSNs, e.g., Deluge and Rateless Deluge, require the radio to be on during the entire reprogramming process. As current sensor nodes (e.g., TelosB and MicaZ) have a similar energy consumption no matter the node is transmitting or receiving. Hence, the energy consumption of reprogramming depends chiefly on the completion time. Nodes can be put into the sleep state by carefully scheduling of the data transmissions, e.g., coordinating among neighboring nodes [22] or setting up a structure for transmission [23]. Sleep scheduling usually incurs extra overheads (in exchanging control-plane messages or in setting up the structure), we do not incorporate it into our current design. We consider it as one direction of our future work.

We also implement ReXOR on the TinyOS/Mica2/TOSSIM platform to investigate the detailed behaviors of propagation. We disseminate a nine-page image with the same page size as in the real experiments. We use the `LossyBuilder` provided in TinyOS to generate a 10-node line topology with 10-feet internode spacing. We investigate the topology and find that nodes 30-feet apart are almost disconnected (e.g., nodes 0 and 3; nodes 1 and 4). Hence, the network topology is at least six hops. Specially, nodes 1 and 8 are at least four hops away.

6.2 Clique Topology

Fig. 7a shows the number of transmitted data packets in the single-hop clique topology consisting a total of five TelosB nodes, for the forced packet loss rate increased from 0 to 30 percent. We can see that ReXOR performs better than Deluge because of data encoding. However, ReXOR performs worse than Rateless Deluge because of link qualities in this topology have low variance (i.e., the

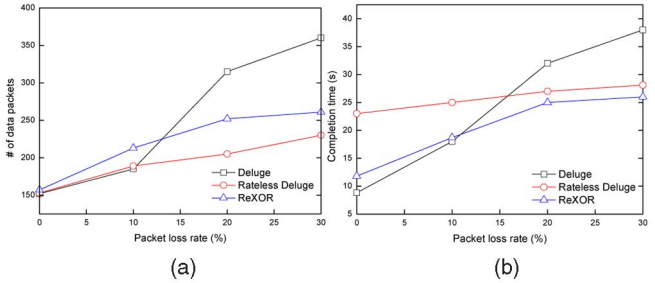

Fig. 7. Data traffic and completion time under the single-hop clique topology consisting of five TelosB nodes. (a) Data traffic. (b) Completion time.

link qualities from the source to other nodes are roughly the same).

Fig. 7b shows the completion times in the single-hop clique topology. When the loss rate is low, the completion time of Rateless Deluge is much longer than both Deluge and ReXOR because of its long decoding time. When the loss rate increases, the completion time of ReXOR is much shorter than Deluge because of XOR encoding. It is even shorter than Rateless Deluge despite it transmits more data packets as illustrated in Fig. 7a. This is because the decoding time of Rateless Deluge is larger than ReXOR.

6.3 Grid Topology

To see how ReXOR performs in multihop sensor networks compared to Deluge and Rateless Deluge, We run these protocols in both the sparse grid topology and the dense grid topology consisting of 16 TelosB nodes.

Fig. 8a shows the number of transmitted data packets in the sparse grid topology. ReXOR transmits far less data packets than Deluge, but transmits a few more packets than Rateless Deluge. Fig. 8b shows the completion times in the sparse grid topology. ReXOR has a shorter completion time than both Deluge and Rateless Deluge because of its lightweight encoding.

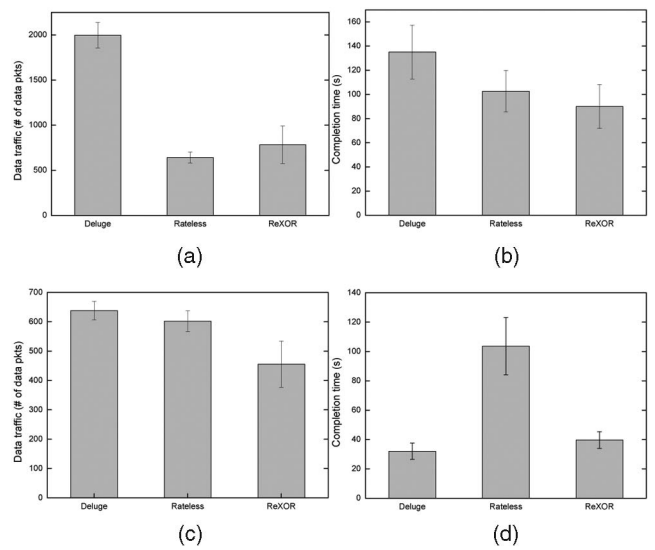

Fig. 8. Data traffic and completion time under the multihop grid topologies consisting of 16 TelosB nodes. (a) Data traffic (sparse grid). (b) Completion time (sparse grid). (c) Data traffic (dense grid). (d) Completion time (dense grid).

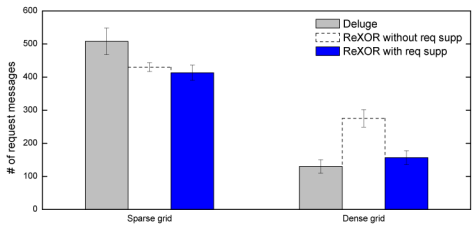

Fig. 9. Request messages in the sparse grid and the dense grid.

This result also implies that ReXOR preserves the high efficiency of Deluge's pipelining strategy for exploiting spatial multiplexing in multihop topologies. To further confirm this point, we use ReXOR to disseminate a nine-page image in a 10-node line topology in TOSSIM [24]. We have found that node 1 completed page 8 at time 101.16 s and node 8 completed page 4 at time 101.21 s. The time for transmitting a 20-packet page is at least $20 \times 20 = 400$ ms (where 20 ms is the packet transmission time in TinyOS for Mica2. Note that TOSSIM simulation model is based on Mica2). This clearly indicates that pages 8 and 4 are transmitted concurrently. Therefore, ReXOR remains the pipelining strategy of Deluge.

Fig. 8c shows the number of transmitted data packets in the dense grid topology. We can see that ReXOR performs better than both Deluge and Rateless Deluge. The performance of Rateless Deluge degrades because of highly variable link qualities. The reason is that in this case the data redundancy of Rateless Deluge is dominated by the poor links. Actually, in the dense case, it makes sense to propagate to nearby neighbors as quickly as possible, thus, further away neighbors can get better link qualities, which, in turn, reduces packet losses and retransmissions. ReXOR achieves a high performance in the dense topology by decreasing its interpage waiting time. Fig. 8d shows the completion time in the dense grid topology. We can see that ReXOR has a much shorter completion time than Rateless Deluge. This further confirms that the importance of ReXOR's density-aware mechanism.

To evaluate ReXOR's probabilistic-based superset suppression mechanism, we compare the number of request messages of Deluge, ReXOR′ (without probabilistic suppression), and ReXOR (with probabilistic suppression) in both sparse and dense grid topologies. Fig. 9 shows the results. Interestingly, we see that both ReXOR′ and ReXOR has fewer request messages than Deluge in the sparse grid topology. This is because ReXOR improves Deluge's performance (in terms of data retransmissions and completion time) in this case, which, in turn, reduces the number of request messages. However, in the dense case, without probabilistic suppression, ReXOR′ increases the request messages by more than 100 percent. With probabilistic suppression (the suppression probability depends on the network density), ReXOR keeps the increase within 20 percent, far less than that of ReXOR′. We continue to examine the actual performance of ReXOR and ReXOR′, with Deluge and Rateless Deluge for references. Table 4 shows the results. In the sparse grid topology, ReXOR has a similar performance as ReXOR′ because few requests will be suppressed in ReXOR if the density is low. In the dense

TABLE 4
Completion Times (s) in the Sparse Grid and the Dense Grid

	Deluge		Rateless Deluge		ReXOR'		ReXOR	
	Avg.	Sd.	Avg.	Sd.	Avg.	Sd.	Avg.	Sd.
Sparse grid	135.0	22.1	102.6	17.2	87	19.7	90	18.1
Dense grid	32.2	5.5	103.6	19.5	46	6.7	39.6	5.7

grid topology, ReXOR has a better performance than ReXOR'. This is because more request messages will be suppressed in ReXOR if the density is high, resulting in fewer message collisions.

7 Related Work

Deluge [2] is perhaps the most popular reprogramming protocol used for reliable code updates in WSNs. It uses a three-way handshake and NACK-based protocol for reliability, and employs segmentation (into pages) and pipelining for spatial multiplexing. It is highly optimized and can achieve one ninth the maximum transmission rate of the radio supported under TinyOS. It was also mentioned by the author of Deluge that the use of FEC improves performance in sparse networks while harming performance in dense networks due to the existence of highly variable link qualities [2]. Other prestigious works include MNP [22], Sprinkler [25], Freshet [26], Stream [27], CORD [23], etc.

MNP [22] provides a detailed sender selection algorithm to choose a local source of the code which can satisfy the maximum number of nodes. Sprinkler [25] uses the localization service at each node to construct a connected dominating set (CDS). It uses TDMA to schedule packet transmissions among the CDS nodes to reduce energy consumption by minimizing packet transmissions. Freshet [26] aggressively optimizes energy consumption during reprogramming. Leveraging a limited topology information, nodes estimate when the code will actually reach their vicinity and enter a sleeping period before that time. Stream [27] reduces the size of data actually disseminated by preinstalling the reprogramming protocol before deployments. CORD [23] is a more recent work whose design objective is to minimize the energy consumption. It employs a two-phase approach in which the object is delivered to a subset of nodes in the network that form a connected dominating set in the first phase, and to the remaining nodes in the second phase. Compared with Sprinkler [25], CORD introduces sleep scheduling at each node to further save energy.

Recently, several coding-based reprogramming protocols specifically designed for WSNs are proposed to address the deficiency of Deluge in sparse and lossy networks, including Rateless Deluge [4], SYNAPSE [5], and AdapCode [6]. They all use network coding to encode a packet before transmission. After receiving an expected number of encoded packets, the receiving node uses Gaussian elimination to decode the packets. The difference is that Rateless Deluge [4] uses Random Linear Codes; SYNAPSE [5] uses Fountain Codes; and AdapCode [6] also uses linear codes, but the coding scheme is adaptively changed according to the link quality.

ReXOR enhances Deluge by employing XOR encoding in the retransmission phase to reduce the communication cost. An important difference between ReXOR and previous coding-based reprogramming protocols [4], [5], [6] is that the (XOR-ed) packets can be instantly decoded at the receiving nodes, eliminating the use of Gaussian elimination. Thus, it is computationally much more lightweight than previous coding-based reprogramming protocols. Another difference between ReXOR and previous coding-based reprogramming protocols is that ReXOR adapts the interpage waiting time in a density-aware manner, retaining a high performance in dense networks.

8 Conclusions

In this paper, we propose ReXOR, a lightweight and density-aware reprogramming protocol for WSNs using XOR. The core idea of ReXOR is 1) leverage the spatial diversity by using a lightweight XOR coding scheme in the retransmission phase to reduce the communication cost and 2) exploit the adaptive behavior by determining the interpage waiting time in a density-aware manner to achieve good performance in both dense and sparse networks.

We have implemented ReXOR based on TinyOS and evaluate its performance extensively. Results show that ReXOR is indeed lightweight compared with previous coding-based reprogramming protocols in terms of computation overhead. The results also show that ReXOR achieves good network-level performance in both dense and sparse networks, compared with Deluge and a typical coding-based reprogramming protocol, Rateless Deluge.

Acknowledgments

The authors thank the anonymous reviewers and the associate editor for their insightful comments. This work was supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, the National Science Foundation of China (Grant No. 61070155), the Program for New Century Excellent Talents in University (NCET-09-0685), and in part by NSERC Discovery Grant 341823-07, NSERC Strategic Grant STPGP 364910-08, and FQRNT Grant 2010-NC-131844.

References

[1] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," *IEEE Network*, vol. 20, no. 3, pp. 48-55, May/June 2006.

[2] J.W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys '04)*, 2004.

[3] TinyOS, http://www.tinyos.net, 2011.

[4] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes," *Proc. ACM/IEEE Int'l Conf. Information Processing in Sensor Networks (IPSN '08)*, 2008.

[5] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F.H. III, and M. Zorzi, "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes," *Proc. Ann. IEEE Comm. Soc. Conf. Sensor, Mesh and Ad Hoc Comm. and Networks (SECON '08)*, 2008.

[6] I.-H. Hou, Y.-E. Tsai, T.F. Abdelzaher, and I. Gupta, "AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks," *Proc. IEEE INFOCOM,* 2008.

[7] L. Gu and J.A. Stankovic, "t-Kernel: Providing Reliable OS Support to Wireless Sensor Networks," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys '06),* 2006.

[8] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "XORs in the Air: Practical Wireless Network Coding," *Proc. ACM SIGCOMM,* 2006.

[9] S. Chachulski, M. Jennings, S. Katti, and D. Katabi, "Trading Structure for Randomness in Wireless Opportunistic Routing," *Proc. ACM SIGCOMM,* 2007.

[10] S. Katti, D. Katabi, H. Balakrishman, and M. Medard, "Symbol-Level Network Coding for Wireless Mesh Networks," *Proc. ACM SIGCOMM,* 2008.

[11] S. Guo, Y. Gu, B. Jiang, and T. He, "Opportunistic Flooding in Low-Duty-Cycle Wireless Sensor Networks with Unreliable Links," *Proc. ACM MobiCom,* 2009.

[12] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *Proc. USENIX First Conf. Symp. Networked Systems Design and Implementation (NSDI '04),* 2004.

[13] S.Y.E. Rouayheb, M.A.R. Chaudhry, and A. Sprintson, "On the Minimum Number of Transmissions in Single-Hop Wireless Coding Networks," *Proc. Information Theory Workshop,* 2007.

[14] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[15] L. Keller, E. Drinea, and C. Fragouli, "Online Broadcasting with Network Coding," *Proc. Workshop Network Coding, Theory and Applications,* 2008.

[16] E. Rozner, A.P. Iyer, Y. Mehta, L. Qiu, and M. Jafry, "ER: Efficient Retransmission Scheme for Wireless LANs," *Proc. ACM Int'l Conf. Emerging Networking Experiments and Technologies,* 2007.

[17] R.A. Costa, D. Munaretto, J. Widmer, and J. Barros, "Informed Network Coding for Minimum Decoding Delay," *Proc. IEEE Int'l Conf. Mobile Ad Hoc and Sensor Systems (MASS '08),* 2008.

[18] C. Fragouli, D. Lun, M. Médard, and P. Pakzad, "On Feedback for Network Coding," *Proc. Ann. Conf. Information Sciences and Systems,* 2007.

[19] H. Seferoglu and A. Markopoulou, "Opportunistic Network Coding for Video Streaming over Wireless," *Proc. Int'l Packet Video Workshop,* 2007.

[20] A. Kamra, V. Misra, J. Feldman, and D. Rubenstein, "Growth Codes: Maximizing Sensor Network Data Persistence," *Proc. ACM SIGCOMM,* 2006.

[21] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, "Performance of Bulk Data Dissemination in Wireless Sensor Networks," *Proc. Fifth IEEE/ACM Int'l Conf. Distributed Computing in Sensor Systems,* 2009.

[22] S.S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS '05),* 2005.

[23] L. Huang and S. Setia, "CORD: Energy-Efficient Reliable Bulk Data Dissemination in Sensor Networks," *Proc. IEEE INFOCOM,* 2008.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," *Proc. ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys '03),* 2003.

[25] V. Naik, A. Arora, P. Sinha, and H. Zhang, "Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices," *Proc. IEEE Int'l Real-Time Systems Symp.,* 2005.

[26] M.D. Krasniewski, R.K. Panta, S. Bagchi, C.-L. Yang, and W.J. Chappell, "Energy-Efficient On-Demand Reprogramming of Large-Scale Sensor Networks," *ACM Trans. Sensor Networks,* vol. 4, no. 1, pp. 1-38, 2008.

[27] R.K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *Proc. IEEE INFOCOM,* 2007.

Wei Dong received the BS degree from the College of Computer Science at Zhejiang University and achieved all credits in the Advanced Class of Engineering Education (ACEE) of Chu Kechen Honors College from Zhejiang University in 2005. He is currently a PhD student in the College of Computer Science of Zhejiang University, under the supervision of Professor Chun Chen. His research interests include networked embedded systems and wireless sensor networks. He is a student member of the IEEE.

Chun Chen received the bachelor of mathematics degree from Xiamen University, China, in 1981 and the MS and PhD degrees in computer science from Zhejiang University, China, in 1984 and 1990, respectively. He is a professor in the College of Computer Science and the director of the Institute of Computer Software at Zhejiang University. His research interests include embedded systems, image processing, computer vision, and CAD/CAM. He is a member of the IEEE.

Xue Liu received the BS degree in mathematics from Tsinghua University, China, the MS degree in automatic control from Tsinghua University, China, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2006. He is currently an associate professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln, where he holds the Samuel R. Thompson Chair professorship in engineering. From 2007-2009, he was an assistant professor in the School of Computer Science at McGill University in Canada. His research interests are in real-time and embedded systems, cyber-physical systems, server and data center performance modeling and management, and software reliability. He worked briefly at Hewlett-Packard Labs and the IBM T.J. Watson Research Center. His work on software reliability received the *IEEE Transactions on Industrial Informatics*' Best Paper award in 2008. He is a member of the IEEE and the ACM.

Jiajun Bu received the BS and PhD degrees in computer science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in the College of Computer Science and the deputy dean of the Department of Digital Media and Network Technology at Zhejiang University. His research interests include embedded systems, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.

Yi Gao received the BEng degree from Zhejiang University in 2009. He is currently a first-year PhD student at Zhejiang University, China. From December 2008 to April 2009, he worked in the Information Systems College of Singapore Management University as an exchange student. His research interests include data reliability and protocols in wireless sensor networks. He is a student member of the IEEE.