Open access • Proceedings Article • DOI:10.1109/INFOCOM.2008.211

AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks

I-Hong Hou, Yu-En Tsai, Tarek Abdelzaher, Indranil Gupta

Institutions: University of Illinois at Urbana–Champaign

Published on: 13 Apr 2008 - International Conference on Computer Communications

Topics: Linear network coding, Wireless sensor network, Wireless network, Network packet and Load balancing (computing)

AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks

I-Hong Hou, Yu-En Tsai, Tarek F. Abdelzaher, Indranil Gupta
Department of Computer Science
University of Illinois, Urbana, IL 61801
{ihou2,ytsai20}@.uiuc.edu, {zaher,indy}@cs.uiuc.edu

***Abstract*—Code updates, such as those for debugging purposes, are frequent and expensive in the early development stages of wireless sensor network applications. We propose AdapCode, a reliable data dissemination protocol that uses adaptive network coding to reduce broadcast traffic in the process of code updates. Packets on every node are coded by linear combination and decoded by Gaussian elimination. The core idea in AdapCode is to adaptively change the coding scheme according to the link quality. Our evaluation shows that AdapCode uses up to 40% less packets than Deluge in large networks. In addition, AdapCode performs much better in terms of load balancing, which prolongs the system lifetime, and has a slightly shorter propagation delay. Finally, we show that network coding is doable on sensor networks in that (i) it imposes only a 3 byte header overhead, (ii) it is easy to find linearly independent packets, and (3) Gaussian elimination needs only 1KB of memory.**

I. Introduction and Motivation

Wireless sensor networks have been widely used to perceive and interact with the physical world for different purposes such as military surveillance [6], habitat monitoring [15], structural monitoring [18], and medical applications [4]. Sensor network applications are typically developed and debugged in the lab then deployed in a representative environment (e.g., outdoors), where the remaining environment-dependent bugs are eliminated. Often, such troubleshooting requires frequent upload of new code, motivating efficient broadcast. The broadcast must be reliable, fast, and minimal in the amount of network bandwidth consumed. Furthermore, it is desired that the load it imposes on the network be balanced in order to balance energy consumption, which reduces the need for battery recharge during field debugging.

Network coding has been recently introduced to reduce traffic in general networks [1]. A lot of work in both wired and wireless networks followed this idea. This reduction of traffic makes the most sense in wireless sensor networks, where nodes have very scarce resources. Moreover, since communication is slow compared to computation, a trade-off between computation and communication can be exploited. It becomes acceptable to do more sophisticated computation in order to reduce the need for transmission. In addition, the broadcast nature of wireless sensor networks increases the benefits of network coding. Due to the one-to-many property, the sink needs to update codes or sends protocol configuration messages to all nodes [7]. We focus on making this broadcast scenario efficient. Encoding packets in intermediate nodes and then sending only coded packet instead of individual packets reduces the traffic and saves energy without increasing delay.

Previous work [8] applying network coding to wireless networks cannot be applied to sensor networks. In sensor networks, memory is so limited that nodes cannot cache overheard packets which might not be useful, and energy is also too precious to broadcast the packets nodes have overheard. A protocol running in sensor networks must be simple and easily implemented. Moreover, the dynamic environment of wireless sensor networks should be considered; nodes can temporarily disconnect or fail and the link quality between nodes can vary over time. A good algorithm should be adaptive to reflect this dynamic nature.

This paper proposes AdapCode, a reliable data dissemination protocol using adaptive network coding, to reduce traffic in the process of code updates. Our coding methodology is to randomly generate N coefficients and compute the linear combination of N packets. Gaussian elimination is used to decode the original packets. From our preliminary work, we found that the best coding scheme (i.e., one that produces the least packets) varies depending on the link density. For example, $N = 8$ is the best when nodes have 12 neighbors, while $N = 2$ is the best when they have only 5 neighbors. Generally, if nodes have more neighbors, they can encode more packets together without losing reliability since they can get enough combinations from their neighbors to decode. Taking advantage of variations in connectivity, we present an adaptive network coding protocol, where nodes dynamically decide N based on how many neighbors they have. Since the broadcast must be received by all nodes, 100% reliability is required and guaranteed by NACK messages. We compare AdapCode with Deluge, a state-of-the-art protocol for propagating new code images, in TinyOS version 2. The results show that AdapCode uses less packets than Deluge does to send an image of the same size in dense deployments. For example, in a large network, AdapCode uses up to 40% less packets to send a code image of 1024 packets and up to 30% less packets to send a code image of 128 packets when nodes have about 7 neighbors on average. Furthermore, AdapCode outperforms Deluge in terms of load balancing. The number of packets sent in AdapCode from the top 10% most-sending nodes is significantly smaller per node than that with Deluge. This property delays the need for changing batteries since the power consumption is more equally distributed across all

nodes in those topologies. Moreover, AdapCode propagates new image of that size within a propagation delay that is up to 15% shorter compared to that of Deluge.

Overheads in AdapCode in our implementation are reasonably low. First, the packet header overhead occurs in two ways; storing coefficients of the linear combination and handling computation overflow. Our analysis shows that storing coefficients needs 17 bits and handling computation overflow needs 5 extra bits in our implementated setting. The total storage overhead is therefore 3 bytes, not significant given the 46-bytes TinyOS packet size. Second, once we randomly choose the coefficients, it is possible to have linearly dependent combinations. This is another form of overhead since it required more packets to be received such that they can be decoded. We show that the expected number of extra packets needed to obtain a given number of linear independent combinations of messages is not large. Finally, we show that the computation cost of Gaussian elimination is only 1KB memory.

The rest of this paper is organized as follows: Section 2 introduces the related work. Section 3 provides some preliminary experimental data that guide algorithm design. The section explores the best coding scheme in different network densities. Section 4 presents our design of AdapCode. We analyze the overhead needed in Section 5. Finally, the evaluation of AdapCode is shown in Section 6.

II. Related Work

Our approach is to apply network coding in sensor networks to reduce the traffic used in propagating large amounts of data. Related research can be divided into three categories: data dissemination, network coding, and network programming.

Power is one of the most critical resources in wireless sensor networks. Since packet transmission is a very energy-consuming action for sensors, a lot of work has focused on reducing packet transmissions. One of the most widely used approaches is to do data aggregation [12]. This approach cannot be used when all the original packets are needed at the received as is the case with propagating code updates. In this paper, we focus on the packet broadcast problem in which all nodes need to receive all the packets. Obviously, naive flooding is also not desirable since it leads to the broadcast storm problem [13]. An approach trying to minimize the number of packets required for a sink to flood queries is probabilistic broadcast [11] [5]. We experimentally show that network coding outperforms probabilistic broadcast.

Network coding [1] is used to improve throughput or save bandwidth. The core idea of network coding is to allow the mixing of data (e.g., by an XOR operation or a linear combination) at intermediate network nodes. Network coding has been applied in general networks. Li et al. applied network coding in wired networks [10]. For wireless networks, Katti et al. take advantage of overheard packets [8]. Their work takes advantage of cross links, which is not feasible in our one-to-many scenario. The approach also incurs overhead since nodes need to broadcast overheard packets they receive. Recently, network coding has been introduced in wireless sensor networks for ubiquitous data collection [3] and continuous data collection [16]. These coding schemes are used for data storage in query-based applications but not for saving bandwidth. Widmer et al. apply network coding in delay-tolerant networks and shows that network coding compares very favorably to probabilistic routing in reliability and robustness [17].

Deluge is now perhaps the most popular data dissemination protocol used for reliable sensor network broadcast [7]. It is heavily used for network code upload [2] [19]. It can disseminate data with 100% reliability at nearly 90 bytes/second. Deluge builds off Trickle, a protocol for maintaining code updates in sensor networks [9]. Trickle uses the epidemic/gossip approach where a node suppresses its own broadcast if it overhears a similar code summary. The advertising rate changes depending on whether the node is up-to-date or not. Trickle can re-program an entire network in as little as twenty seconds. Trickle only provides a mechanism for determining when nodes should propagate code and only deals with single packets. Deluge adds full support for the dissemination of large data objects. Deluge has been integrated into TinyOS and was recently ported to TinyOS version 2. We aim at further reduce the traffic needed in the network programming process.

III. Preliminary Exploration

Network coding helps in sensor networks for two main reasons. First, sensor nodes are extremely resource restricted and hence traffic needs to be minimized. One may be willing to pay the cost of computation in order to reduce traffic. Second, sensor networks have a broadcast nature. Previous research applying network coding in wireless networks focused on unicast since there are not many broadcast scenarios in wireless networks. In contrast, we study the case where some information (e.g., code updates [7] or network reconfiguration), needs to be known by all nodes.

Our work tries to use network coding to reduce the traffic of code dissemination. A simple example of how network coding reduces traffic in such a broadcast scenario can be seen in Figure1. The sink, A, needs to broadcast 2 packets, a and b. If nodes simply forward the messages they receive, A, B, and C need to send a total of 6 packets (2 packets each). With network coding, B and C can a combination of a and b, say $a + b$ and $a + 2b$ respectively. D and E can then simply decode the packets by solving linear equations. We thus save two packets in total.

Our methodology of coding is to combine n packets into one coded packet using linear combination. In node t, we generate n coefficients $a_{t,1} \sim a_{t,n}$ and compute $b_t = a_{t,1} \cdot p_1 + a_{t,2} \cdot p_2 + \ldots + a_{t,n} \cdot p_n$, where p_k is the kth packet and b_t is the coded packet. The n coefficients are included in the header of the coded packet. Therefore, a node receiving n coded packets can easily solve $Ap = b$ by Gaussian elimination.

In the following two subsections, we present a preliminary performance exploration of naiive network coding that motivates AdapCode design. First, we show by simulation that the choice of the optimal number of packets to combine into one

Category	Parameter	Value
Channel	PATH_LOSS_EXPONENT	4.0
	SHADOWING_STANDARD_DEVIATION	4.0 dB
	PL_D0	55.0 dB
Radio	NOISE_FLOOR	-105.0 dBm
	WHITE_GAUSSIAN_NOISE	4

TABLE I: Parameter setting of our simulations

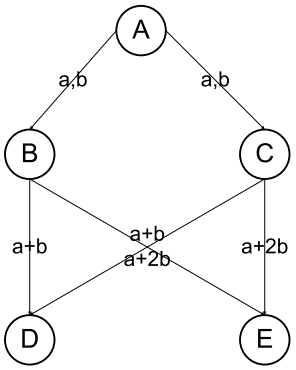


Fig. 1: The network coding diagram.

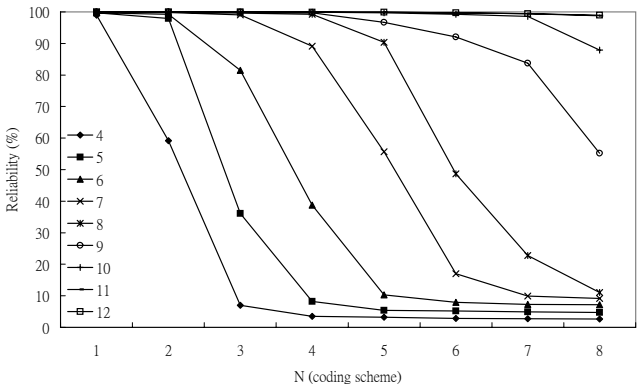


Fig. 2: Each curve shows the reliability of different coding schemes in a specific density. The density is shown in terms of number of neighbors.

depends on network density. We quantify this dependency and exploit it later in our adaptive algorithm.

Second, given a well-tuned scheme (that combines, say, every N packets), we show that naiive network coding outperforms a store-and-forward approach given the same bandwidth consumption. Observe that, in the network coding scheme, a node sends one message for every N messages it receives. In other words, compared to naiive flooding, the bandwidth saved is up to $\frac{N-1}{N}$. Another active approach saving an equal amount of bandwidth is probabilistic forwarding, where a node re-broadcasts an incoming message with probability $\frac{1}{N}$. Comparing the two schemes gives some preliminary insight into the raw advantages of network coding.

We performed our preliminary experimental exploration using the TOSSIM simulator [14]; a standard tool in sensor network simulation that runs actual network code on simulated nodes. The parameter settings are shown in Table I. We considered a 10×10 grid of nodes. In the simulated deployment, there is a source in the network that keeps on sending broadcast messages. The node density is represented by the average number of neighbors, which is varied from 4 to 12, per node.

A. A Motivation for Adaptive Coding

When performing network coding, nodes cannot decode packets until they can get enough combinations to decode. If a sufficient number of independent combinations is not received, reliability is lost. Hence, it is interesting to explore the reliability implications of chosing the number of packet to combine. We refer to a particular choice of such number by *coding scheme*. Hence, it is interesting to compare the reliability of different coding schemes in a given network.

To compute reliability, we ran different coding schemes on at different network densities. In each scheme, we specify N, the number of packets to be combined. In other words, each node will send out one packet containing a linear combination of N messages upon receiving enough data. Therefore, the number of packets sent using network coding is $\frac{1}{N}$ of that of naiive flooding. Obviously, a node cannot decode any message until it receives N packets. We compute *reliability*, defined as the fraction of nodes that can successfully decode all the messages. The result is shown in Figure 2. As intuition, reliability will drop as N increases and density decreases because it becomes harder to receive enough packets to decode data successfully. However, most nodes can still decode all messages under network coding as long as the node density is high enough. For example, when nodes have 12 neighbors, more than 98% of them can decode all messages when N is 8. In a sparse scenario, where nodes only have 5 neighbors, the reliability remains 97% if we reduce N to 2 (which means we can save 50% traffic without considering NACKs and retransmission). This result clearly suggests that network coding can reduce traffic without significant loss on reliability, but N must be adapted to network connectivity.

In the ultimate implementation, we need 100% reliability. Therefore, we add Negative-ACK (NACK) to the naiive scheme. Observe that without the NACKs, a node receiving $N-1$ packets and a node receiving no packets at all are equally problematic because neither of them can decode any of the original N messages. When NACKs are used, nodes receiving less than N packets can send out NACKs to retrieve missing

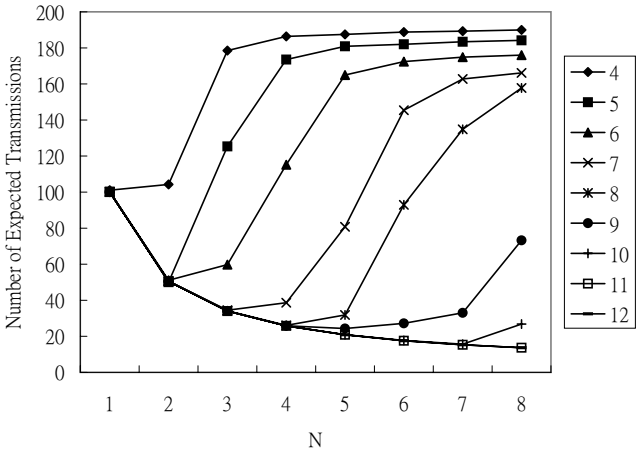


Fig. 3: Number of expected transmissions per packet for different node densities. The density is shown in terms of number of neighbors.

data they need that has already been decoded elsewhere. In the above example, a node receiving $N-1$ packets will need only one more packets to decode all the messages.

To reflect the effect of NACKs on performance, we measure another metric, which we call the *number of expected transmissions*. The number of expected transmissions is defined as the total number of transmissions plus twice the number of missing packets. The intuition behind the definition is that once a node misses a packet, it must send out one NACK to one of its neighbors. The neighbor will reply with the needed packet. This procedure results in two extra transmissions. Although the number of expected transmissions is only a rough estimate, it can serve as a guideline on how to choose a reasonable N. Figure 3 depicts the number of expected transmissions per message sent by source under different node densities and coding schemes. Note that, there are always 100 nodes in our topologies. Hence, without network coding, at least 100 messages are incurred when the source wants to broadcast a packet. Using this figure, we can obtain the coding scheme that can result in the fewest number of expected transmissions for topologies with different grid size. For example, when nodes have 8 neighbors, setting $N = 4$ can achieve the minimum number of expected transmissions.

This helps us understand how good network coding can be in different densities. Coding lots of packets together helps reduce the traffic, but it also decreases reliability. Once the reliability drops, we need more NACK/reply traffic to alleviate the situation. The number of expected packets enables us to balance the tradeoff between traffic and reliability. The best coding scheme for various densities is shown in Table II to demonstrate how network coding is using up redundant links.

B. Network Coding versus Forwarding

Next, we compare naiive network coding (with no NACKs) to forwarding for the same bandwidth consumption. As alluded above, to ensure same bandwidth consumption, the forwarding

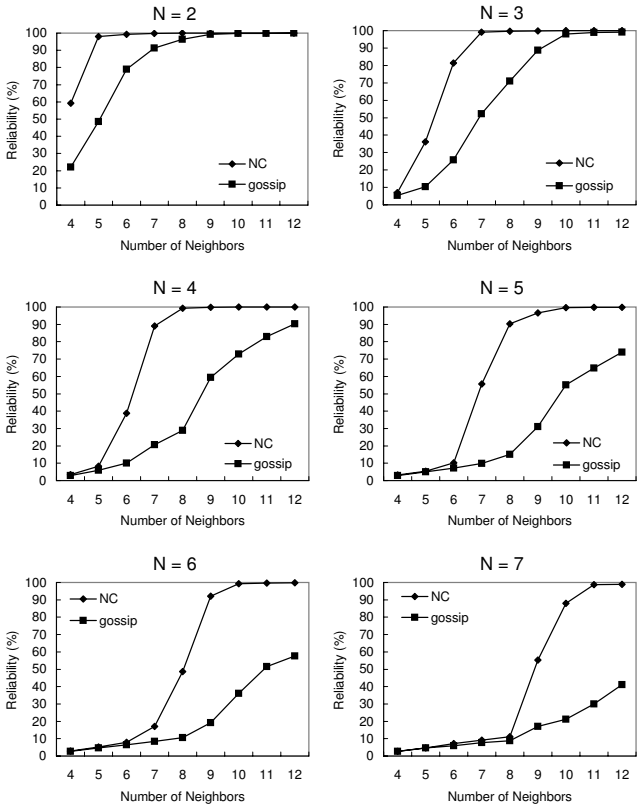


Fig. 4: Reliability of network coding(NC) and gossip

scheme must forward received packets with a probability of $1/N$ (i.e., it is a gossip scheme) whereas the network coding scheme reduces every N received packets into one forwarded packet, ensuring the same receive to forward ratio. The results of this comparison for different N are shown in Figure 4. In most cases, network coding has better reliability than gossip. For example, when $N = 6$, the reliability of network coding remains 99% when nodes have 10 neighbors. Meanwhile, the reliability of gossip becomes 36% under the same setting. Network coding is therefore preferable in the sense that it will require less NACK overhead to ensure reliable broadcast.

This preliminary experiment suggests network coding is a better way to disseminate data across the whole network. It is not difficult to see why gossip behaves so poorly. When we use gossip, a node randomly decides whether to forward a received packet or not. It is very likely for two neighboring nodes to forward the same packet, resulting in useless duplicate packets. Network coding, on the other hand, sends out a random linear combination of all the N packets. It is less probable for two nodes to choose the same (or a dependent) linear combination. Therefore, useless duplicate packets occur less frequently when we use network coding.

IV. AdapCode Design

The results of the previous section suggest that we can reduce traffic by applying network coding and if the coding

Avg. number of neighbors	12	11	10	9	8	7	6	5	4
Best coding scheme	N=8	N=8	N=7	N=5	N=4	N=3	N=2	N=2	N=1

TABLE II: Best coding scheme derived from simulation

scheme is adaptive to network conectivity, the overhead of reliable broadcast can be kept small. We next summarize the properties of a code distribution protocol that we aim to satisfy:

- **Reliability:** When performing code distribution, every node in the network should correctly receive the updated code. This is accomplished by using NACKs. In this paper, we do not address temporary node failures and reboots.
- **Adaptation:** Although Table II has shown us the best coding scheme under different connectivity, in a real deployment, node densities may not be known a priori. Also, since nodes may run out of energy or suffer changes in radio range, node connectivity may not remain static. Therefore, a protocol using network coding should be able to decide its coding scheme adaptively using each node's local knowledge.
- **Low Memory Usage:** Memory is a limited and valuable resource in sensors. A MicaZ sensor typically has only 4 kilobytes of RAM. Hence, a protocol using network coding should not require large memory usage (e.g., for doing Gaussian elimination). Observe that any memory used by dissemination protocol can be released after code distribution finishes. Therefore, doing network coding during the code distribution period will not interfere with sensor operations that follow such distribution. Nevertheless, it is still vital that our protocol can fit into the RAM of sensors.
- **Rapid Propagation:** When a developer wants to do code distribution frequently, a desired feature is that the updated image is propagated to every node in the network rapidly. The time taken to propagate the updated image should be of the order of seconds.
- **Low Traffic:** The traffic required by code distribution should be very small. If too much traffic is introduced, the code distribution procedure may use up too much energy and hurt battery life.
- **Load Balancing:** The traffic sent by each sensor should be approximately balanced. If a small portion of sensors incur a significantly larger amount of traffic than others, those heavy-loaded sensors may fail much quicker.

Next, we describe a protocol that achieves the above performance goals.

A. Protocol Overview

In our protocol, we assume there are n data messages, each with fixed length that can fit into a packet. There is one single source in the system. The source will keep sending packets containing those messages. All the other nodes will help spread messages they receive. Those nodes will use network coding to minimize the number of transmission while ensuring that every active node in the system will correctly receive those messages.

We divide the messages into sequentially numbered *pages*. Each page contains a fixed number M of messages. We explicitly require M to be a power of 2. In our system, we choose M equals 8. The source will keep on transmitting packets and will pause for a period of T milliseconds after finishing a page. This pause of source is necessary to allow other nodes to start propagate previous pages. The choice of T is a tradeoff between traffic and propagation time, which we will discuss in section VI. When a node receives a packet, it first runs Gaussian elimination to see if it has gathered enough information to decode all messages in the packet's page. When it succeeds in decoding all messages within a page, it determines its coding scheme, N, according to the number of its neighbors. We require N to be a factor of M. This also implies that N must be a power of 2. After determining the coding scheme, the node sends out $\frac{M}{N}$ packets, each containing a linear combination of N messages in the page. The coefficients of each linear combination are randomly chosen from 0 to $p-1$, where p is a prime number. Furthermore, we make the leading coefficient of every linear combination be 1. In the implementation, we choose p to equal 5. Also, to avoid multiple sensors transmitting at the same time, which can cause serious collisions, sensors will randomly backoff for a short time before they try to transmit. In our design, the period of backoff is uniformly chosen between 10ms and 74ms. After finishing transmitting those packets, the sensor puts the messages it just decoded into program memory.

B. Adaptively Determining Coding Scheme

As mentioned above, we determine the coding scheme by the number of neighbors that a sensor has. When a sensor starts to buffer packets for a page, it keeps a counter, $curNeighbor$. This counter is defined as the number of different sources of the packets that the sensor knows of. After the sensor succeeds in decoding that page, it computes its long-term number of neighbors, $avgNeighbor$, using the formula: $avgNeighbor = \alpha \times avgNeighbor + (1 - \alpha) \times curNeighbor$. The value of α should be determined according to the stability of the network. For example, if nodes in the network fail frequently, we should have a small value of α to be resilient against topology changes. On the contrary, if the topology remains quite static over time, we should set α to a larger value to obtain a more accurate number. In our implementation, we choose $\alpha = \frac{2}{3}$.

The sensor then decides N according to $avgNeighbor$ and Table II. From Table II, we can obtain a set of 2-tuples with the form (a, b), where a is the average number of neighbors and b

is the best coding scheme. The decision on N is made as the largest power of 2 such that there exists a 2-tuple (a, b) with $a \geq avgNeighbor$ and $b \leq N$. Table III shows the resulting decision derived from the above procedure.

C. Dealing with NACKs

During the code update process, every node keeps a countdown timer. A node will send out a NACK to the local broadcast address when the timer fires. In the NACK packet, the sensor will indicate the page number it is asking for and messages it needs to decode all messages in the page. Since the source will pause for T milliseconds between pages, the delay between pages is at least Tms. The value of the timer is initially set to $2T$ milliseconds.

A key problem in dealing with NACKs is to determine which node should respond to the NACK. If multiple nodes respond simultaneously, they not only incur unnecessary transmissions but may also cause serious packet collisions and channel congestion. To solve the problem, we design a mechanism to distributively select the responder to the NACK without any maintenance overhead. When a node receives a NACK message, it first checks whether it can reply with the needed data. If it can do so, the node will delay for a random period of time to see if any of its neighbors is replying to this NACK. If no reply is heard before the timeout, this node will respond to the NACK. Using this mechanism, we can significantly reduce the risk of simultaneous responses to NACKs.

Further, we adopt a "lazy NACK" mechanism to reduce the number of NACKs. When a node sends out a NACK, it doubles the value of its countdown timer. The value of the timer will be restored to T once the node receives a packet in its page. Also, if a node overhears a NACK containing the same page number as its own, it will reset its timeout timer to avoid sending duplicate NACK.

The detailed algorithm of AdapCode design is shown as follows.

Algorithm 1 ADAPTCODE

```
coeffMatrix ← a M × M matrix
invMatrix ← a M × M matrix
curNeighbor ← 0
timerValue ← 2T
set countdown timer equals timerValue ms
while code distribution is going on do
  if a packet is received then
    sender ← the sender of the packet
    pageNumber ← the page number of the packet
    if sender has not been seen before then
      curNeighbor ← curNeighbor + 1
    timerValue ← 2T
    set countdown timer equals timerValue ms
    construct coeffMatrix using coefficients from
    packets in page pageNumber
    rank ← Gaussian(coeffMatrix, invMatrix)
    if rank = M then
      solve all messages in the page by invMatrix
      avgNeighbor = α × avgNeighbor + (1 − α) ×
      curNeighbor
      determine N using avgNeighbor
      broadcast M/N packets
      curNeighbor ← 0
      pageNumber ← pageNumber + 1
  if a NACK is received then
    pageNumber ← the page number of the packet
    if page pageNumber is already received then
      wait for a random period
      if no response heard during the period then
        reply to the NACK
  if timer timeouts then
    send a NACK
    timerValue ← 2 × timerValue
    set countdown timer equals timerValue ms
```

Algorithm 2 Gaussian($coeffMatrix, invMatrix$)

```
run Gaussian elimination on coeffMatrix
if coeffMatrix is invertible then
  invMatrix ← the inverse matrix of coeffMatrix
return the rank of coeffMatrix
```

V. COST ANALYSIS

Wireless sensors have very limited memory and transmission ability. Many widely used techniques for wireless networks can not be carried out on sensor networks since their costs are too high for the capacities of wireless sensors. In this section, we study the cost of our network coding protocol.

A. Packet Overhead

When a node transmits a packet containing a linear combination of messages, it needs to put the coefficients it chooses in the packet. This will induce additional overhead. A packet can be composed by at most M messages. Since we make the leading coefficient be 1, there are at most $M-1$ coefficients left to be specified. These coefficients can be any integer between 0 and $p-1$. Therefore, there are p^{M-1} choices for the coefficients, which need at most $\lceil \log_2 p^{M-1} \rceil$ bits to specify.

Overhead is also induced by the possibility of overflow when doing linear combination. Suppose each message has t bits. Its numeric value will be no larger than $2^t - 1$. The linear combination involves at most M messages, each with coefficient at most $p-1$. Hence, the numeric value of the linear combination is at most $(p-1) \times M \times (2^t - 1)$. The number of bits needed to represent this value is $\lceil \log_2(p-1) \times M \rceil + t$. Overhead caused by overflow is hence $\lceil \log_2(p-1) \times M \rceil$.

In our implementation, we choose M to be 8 and p to be 5. The overhead caused by coefficients is 17 bits. The overhead caused by overflow is 5 bits. Hence, the overall overhead for our protocol is at most 22 bits, which is less than 3 bytes. Compared to a standard MicaZ packet, which has 46 bytes, the overhead is acceptably small.

avgNeighbor	0 – 5	5 – 8	8 – 11	11 –
N	1	2	4	8

TABLE III: The choice of N according to $avgNeighbor$

B. Solvability of Gaussian Elimination

Ideally, we need M different linear combinations of messages to yield the original messages in a page. However, since we limit the choice of coefficients to integers between 0 and $p-1$, it is possible that some linear combinations are linear dependent. In this section, we derive an upper bound on the expected number of packets needed to obtain M linear independent combinations of messages.

We treat the coefficients of every linear combination as a vector with length M. Our goal is to derive the number of vectors needed to obtain M linearly independent vectors, given that every entry, except the first one, in a vector are randomly chosen between 0 and $p-1$ and the first entry is 1. For the ease of analysis, we consider the linear dependency in Z_p^M, the residue class of modulo p. In Z_p, all operations are treated as modulo operations. For example, in Z_5, we have $3+3=1$ and $2\times 3=1$. Since every set of linearly dependent vectors is also linearly dependent in Z_p^M, our simplification will yield the upper bound of the expected number of vectors needed.

Let E_i, $1 \le i \le M$, be the expected number of vectors needed to obtain i linear independent vectors in Z_p^M. Our goal is to derive the value of E_M. Obviously, $E_1 = 1$. Suppose we already have j linear independent vectors, namely, $v_1, v_2, \cdots, v_j$. Now we are given another randomly generated vector, v_{j+1}, and we wish to compute the probability that v_{j+1} is independent from $v_1, v_2, \cdots, v_j$. If v_{j+1} is not independent from those vectors, there exist constants $c_1, c_2, \cdots, c_j$ such that $\sum_{k=1}^{j} c_k \times v_k = v_{j+1}$. Now, note that the first entries of all these vectors are 1. Hence we have $\sum_{k=1}^{j} c_k = 1$. The number of different choices of $(c_1, c_2, \cdots, c_k)$ satisfying this constraint is p^{j-1}. This implies there are p^{j-1} different vectors that are linearly dependent with $v_1, v_2, \cdots, v_j$. Since there are p^{M-1} different choices for v_{j+1}. The probability that v_{j+1} is linearly independent from the other vectors is $1 - \frac{p^j}{p^M}$. In other words, after obtaining j linearly independent vectors, we need, on average, $\frac{p^M}{p^M - p^j}$ more vectors to yield another linearly independent vector. This results in the recursion formula: $E_{i+1} = E_i + \frac{p^M}{p^M - p^j}$, for all $i \ge 1$. Solving this formula we obtain $E_M = 1 + \sum_{j=1}^{M-1} \frac{p^M}{p^M - p^j}$. In our implementation settings, we have $E_M = E_8 = 8.30$, which is very close to 8, the least number of packets needed to decode all messages in one page.

C. Feasibility of Gaussian Elimination

Wireless sensors are known for having very limited memory. It is very important that Gaussian elimination does not use up too much memory. In this section, we study the memory needed to run Gaussian elimination.

Typically, Gaussian elimination requires two $M \times M$ matrixes, one to store the original coefficients and the other to store the inverse matrix. In our implementation, we store numerators and denominator in different places, resulting in four $M \times M$ matrixes. There are totally $4 \times M^2$ elements in those matrixes. What's left is to determine the size of each element.

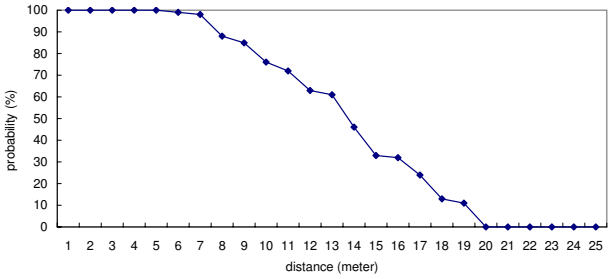


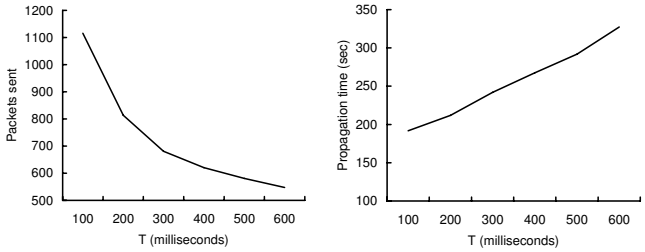
Fig. 5: The probability of receiving a packet from another node in different distances

Let $A = [a_{ij}]$ be the coefficient matrix composed of coefficients in received packets and A_{ij} be the matrix obtained from A by deleting row i and column j. Also, let $[A]$ be the determinant of matrix A. Every element in the inverse matrix A^{-1} will be in the form of $\frac{[A_{ij}]}{[A]}$. Therefore, the size of elements in the matrixes should be big enough to hold the maximum possible value of $[A]$. Let D_M be the maximum possible value of determinant of the coefficient matrix. Note that $D_1 = 1$ since we require the leading coefficient to be 1. According to the definition of determinant, $[A] = \sum_{i=1}^{M} a_{i1} \times [A_{i1}]$, for all $M > 1$. Now that a_{i1} is at most $p-1$ and $[A_{i1}]$ is no larger than D_{M-1}, the value of $[A]$ is upper bounded by $M \times (p-1) \times D_{M-1}$ and we can obtain the recursion: $D_M \le M \times (p-1) \times D_{M-1}$. Solving the inequality yields: $D_M \le M! \times (p-1)^{M-1}$. In our implementation settings, D_M is less than 2^{31}, meaning that we only need to allocate 4 bytes for each element in the matrixes. Thus, the total memory usage of Gaussian elimination would be $4 \times 4 \times M^2 = 2^{10}$bytes $= 1$KB, which can fit in the memory of most modern sensors. In our implementation, AdapCode requires 1433 bytes in RAM.

VI. Performance Evaluation

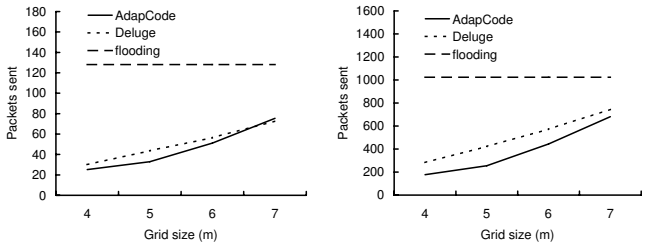
A. Simulation Settings

In this section, we present our simulation results for AdapCode. We consider a 10×10 grid of MICAZ nodes simulated in TOSSIM of TinyOS version 2. Table I shows the simulation settings. To have an idea about what the transmission range is in this parameter setting, we show the probability of receiving a packet from another node in Figure5.



(a) Traffic (b) Latency

Fig. 6: Performance of AdapCode under different T



(a) 128 data segments (b) 1024 data segments

Fig. 7: Number of packets sent per node for different grid sizes

B. Interpage Pause Interval

As mentioned in section IV, the source will pause for T milliseconds after transmitting a page. The choice of T is a tradeoff between traffic and latency. Obviously, larger T will result in longer latency. On the other hand, large T can help reduce traffic due to two reasons: First, larger T implies larger timeout intervals before a node sends out a NACK. The number of unnecessary NACKs is hence reduced. Moreover, since nodes decide their coding scheme based on the number of neighbors heard in a page, larger T will allow nodes have enough time to make a good estimation on their number of neighbors. This will enable nodes to choose a coding scheme that can incur the least traffic.

To show the influence of T on the performance of AdapCode, we measure both the mean number of packets sent per node and the time needed to disseminate an image with 1024 packets. As shown in Figure 6b, the time needed to disseminate the image almost grows linearly as T increases. On the other hand, Figure 6a shows that increasing T reaches deminishing returns in terms of reducing traffic. For example, when we change T from 100 to 200, the mean number of packets sent per node drops from 1115 to 813, resulting in a 302 packet reduction. However, when we change T from 500 to 600, the reduction in traffic is merely 34 packets. In our design, we choose $T = 300$ since this results in both low traffic and low latency.

C. Traffic

In the following sections, we compare AdapCode with Deluge, the state-of-art code dissemination protocol. We evaluate these two protocols using three metrics: traffic, load balance, and propagation delay. We assume that the source needs to broadcast a piece of code that can be divided into D packets. We run both protocols 50 times in a grid deployment for each grid size between $4m$ and $7m$. To see how AdapCode behaves under different code sizes, we evaluate the performance for both $D = 128$ and $D = 1024$, which approximately corresponds to code images with sizes 2KB and 20KB.

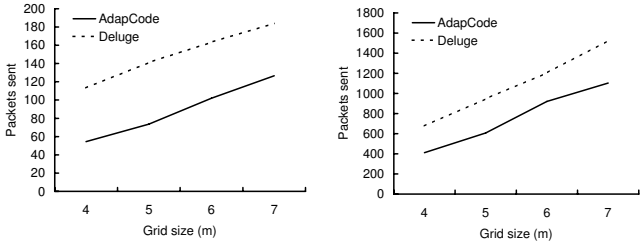
We first compare the mean number of packets per node needed to broadcast D data packets. In addition to data packets, we also count the number of NACKs and replies to NACKs in AdapCode. Similarly, since Deluge uses epidemic messages and request messages to trigger code dissemination, we count the number of these two types of packets. Further, we include naiive flooding as a performance baseline. In flooding, every node needs to broadcast whatever messages it receives. The mean number of packets per node needed for flooding is computed as the number of data segments, which is 128 and 1024 for different settings on D. In other words, we assume there is no packet loss and retransmission in flooding. The mean number of packets sent per node for different grid sizes are shown in Figure 7.

It is very clear from the figure that AdapCode uses significantly fewer packets than flooding for all grid length sizes. Although we unfairly assume links in flooding never drop a packet, AdapCode uses 41–80% and 33–83% fewer packets than flooding for $D = 128$ and $D = 1024$, respectively. AdapCode also has better performance when compared with Deluge. The performance gain is greater when the grid size is small. When $D = 128$, AdapCode uses up to 24% less traffic than Deluge does. When $D = 1024$, the save of traffic by AdapCode is even more significant, up to 40%, than that by Deluge. These results prove that our protocol is adaptive enough to choose a coding scheme that can reduce traffic without incurring too many retransmissions.

D. Load Balancing

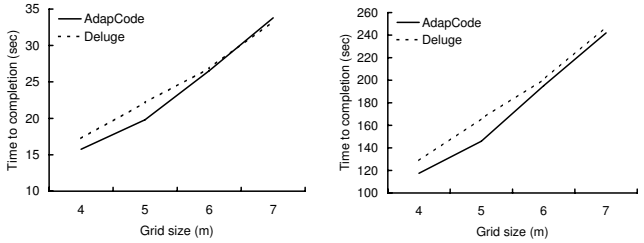
The main motivation for saving traffic is to reduce the energy consumed and to prolong the lifetime of the network. Therefore, load balancing is almost as important as bandwidth usage efficiency. If some nodes are too heavily loaded, those nodes will tend to die out quickly, which can potentially influence the connectivity and coverage of the network. To compare the quality of load balancing of AdapCode and Deluge, we compute the number of packets sent by the 10% nodes that send the most packets. Figure 8 shows the average number of packets transmitted per such node. The results show that AdapCode achieves a much better load balance than Deluge. Moreover, the differences between AdapCode and Deluge are not severely influenced by grid sizes. When $D = 128$, the heavy weighted nodes in AdapCode transmit 57–67 fewer packets than those nodes in Deluge. When $D = 1024$, the differences become 266–422 packets. Therefore, AdapCode can do a better job in prolonging network lifetime by achieving good load balance.

The reason why AdapCode has better load balancing properties is because AdapCode has better function distribution



(a) 128 data segments (b) 1024 data segments

Fig. 8: Mean packets transmitted per node from the 10% heaviest weighted nodes



(a) 128 data segments (b) 1024 data segments

Fig. 9: Mean time required for all nodes to receive all data

properties. Every node in AdapCode needs to forward some data packets once it decodes messages in a page. Hence, the load of forwarding messages is equally distributed among neighbors. Further, when a node receives a NACK, it needs to backoff for a random period of time before it replies to the NACK. This mechanism makes every node have similar probability of replying to NACKs. Further, since a page in AdapCode consists of only 8 messages, the burden of replying to a single NACK is small.

E. Propagation Delay

Another important metric in code dissemination protocols is the time taken to disseminate the code. Since developers may usually want to frequently update code in the debugging stages, high propagation delay will make the procedure painfully slow. From Figure 9, we can see that AdapCode satisfies the requirement of small propagation delay. AdapCode generally takes less time than Deluge to complete code dissemination. When $D = 128$, AdapCode is quicker than Deluge by about 10Furthermore, AdapCode is always quicker than Deluge when $D = 1024$. The latency difference between the two protocols can be as high as 20 seconds, or 15%. Since Deluge is known to be a highly optimized code dissemination protocol, this result shows AdapCode has much promise. We have not yet optimized its implementation but have improved over Deluge nevertheless.

VII. Conclusion

In this paper, we present AdapCode, a code dissemination protocol that achieves low traffic, low latency, and good load balancing. The core idea of AdapCode is to (i) take advantage of redundant links in wireless sensor networks by using network coding to reduce data packets, and (ii) exploit adaptive behavior to choose the best coding scheme to reduce NACK/reply packets.

We analyze the cost of network coding and conclude that network coding is feasible on wireless sensors in terms of overhead and memory size. We then implement AdapCode and compare it against Deluge, the most widely used code dissemination protocol. We observe that AdapCode outperforms Deluge in all three important performance metrics: traffic, load balancing, and latency.

References

[1] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[2] A. Arora, R. Ramnath, and E. Ertin. Exscal: Elements of an extreme scale wireless sensor network, 2005.

[3] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *Proc. of IPSN '05*, 2005.

[4] R. Ganti, P. Jayachandran, T. F. Abdelzaher, and J. A. Stankovic. Satire: a software architecture for smart attire. In *Proc. of ACM MobiSys '06*, 2006.

[5] Z. Haas, J. Halpern, and L. Li. Gossip-based ad hoc routing. In *Proc. of IEEE INFOCOM'02*, 2002.

[6] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-efficient surveillance system using wireless sensor networks. In *Proc. of ACM MobiSys '04*, 2004.

[7] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of ACM SenSys '04*, 2004.

[8] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft. Xors in the air: Practical wireless network coding. In *Proc. of ACM SIGCOMM'06*, September 2006.

[9] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of USENIX NSDI '04*, 2004.

[10] Z. Li and B. Li. Network coding in undirected networks. In *Proc. of the 38th Annual Conference on Information Sciences and Systems (CISS)*, 2004.

[11] J. Luo, P. Eugster, and J.-P. Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networks. In *Proc. of IEEE INFOCOM'02*, 2003.

[12] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of USENIX OSDI*, December 2002.

[13] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proc. of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 1999.

[14] M. Welsh D. Culler P. Levis, N. Lee. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proc. of ACM SenSys '03*, 2003.

[15] G. Tolle, J. Polastre, R. Szewczyk, N. Turner, K. Tu, S. Burgess, D. Gay, P. Buonadonna, W. Hong, T. Dawson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. of ACM SenSys '05*, 2005.

[16] D. Wang, Q. Zhang, and J. Liu. Partial network coding: Theory and application for continuous sensor data collection. *Proc. of IEEE IWQoS*, June 2006.

[17] J. Widmer and J. Boudec. Network coding for efficient communication in extreme networks. In *Proc. of ACM SIGCOMM WorkShop WTDN'05*, August 2005.

[18] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. of ACM SenSys '04*, 2004.

[19] Z. Yao, Z. Lu, H. Marquardt, G. Fuchs, S. Truchat, and F. Dressler. On-demand Software Management in Sensor Networks using Profiling Techniques. In *ACM Second International Workshop on Multi-hop Ad Hoc Networks: from theory to reality 2006 (ACM REALMAN 2006), Demo Session*, pages 113–115, May 2006.