SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes

Michele Rossi, *Member*, *IEEE*, Nicola Bui, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, *Student Member*, *IEEE*, and Michele Zorzi, *Fellow*, *IEEE*

Abstract—This paper presents SYNAPSE++, a system for over the air reprogramming of wireless sensor networks (WSNs). In contrast to previous solutions, which implement plain negative acknowledgment-based ARQ strategies, SYNAPSE++ adopts a more sophisticated error recovery approach exploiting rateless fountain codes (FCs). This allows it to scale considerably better in dense networks and to better cope with noisy environments. In order to speed up the decoding process and decrease its computational complexity, we engineered the FC encoding distribution through an original genetic optimization approach. Furthermore, novel channel access and pipelining techniques have been jointly designed so as to fully exploit the benefits of fountain codes, mitigate the hidden terminal problem and reduce the number of collisions. All of this makes it possible for SYNAPSE++ to recover data over multiple hops through overhearing by limiting, as much as possible, the number of explicit retransmissions. We finally created new bootloader and memory management modules so that SYNAPSE++ could disseminate and load program images written using any language. At the end of this paper, the effectiveness of SYNAPSE++ is demonstrated through experimental results over actual multihop deployments, and its performance is compared with that of Deluge, the de facto standard protocol for code dissemination in WSNs. The TinyOS 2 code of SYNAPSE++ is available at http://dgt.dei.unipd.it/download.

Index Terms—Wireless sensor networks, distributed networks, data communications, protocol architecture, protocol verification, error control codes, system integration and implementation.

1 Introduction

Wireless reprogramming is an invaluable service for wireless sensor networks (WSNs). Code updates are in fact essential to reconfigure the network on the fly after, e.g., a topology change, fix bugs in the software, re-task or update existing applications. This service needs to be fully reliable, scalable, energy efficient, and rapid, as the time spent with the radio on is the main source of energy consumption in WSNs. The program, due to the memory limitations of the nodes (usually 10 kB of RAM), should be split into blocks and processed one block at a time. This affects the design of dissemination and error control algorithms. Further, the code should be efficiently propagated over multihop networks without a priori knowledge of the topology. Lastly, WSNs are usually highly populated with devices, thus if no proper countermeasures are taken, it is likely that many senders will transmit at the same time. The downside of this is that packets will collide, thus resulting in an overall degradation of the performance in terms of reprogramming time and energy efficiency.

- *M. Rossi, G. Zanca, and M. Zorzi are with the Department of Information Engineering, University of Padova, Via Gradenigo 6/B, 35131 Padova, Italy. E-mail: {rossi, zancagio, zorzi}@dei.unipd.it.*
- *N. Bui is with the Consorzio Ferrara Ricerche, Via Saragat 1, Blocco B, 44124 Ferrara, Italy. E-mail: bui@dei.unipd.it.*
- *L. Stabellini is with the Wireless@KTH, Royal Institute of Technology, Electrum 418, SE-164 40 Kista, Sweden. E-mail: lucast@kth.se.*
- *R. Crepaldi is with the Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801. E-mail: rcrepal2@uiuc.edu.*

Manuscript received 5 Nov. 2008; revised 5 Jan. 2009; accepted 7 Aug. 2009; published online 3 June 2010.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2008-11-0446.
Digital Object Identifier no. 10.1109/TMC.2010.109.

Classical schemes for reprogramming WSNs [1], [2], [3], [4] use sophisticated error recovery algorithms that rely on a selective NACK-based retransmission approach. While these intelligently handle the selection of senders and feature special mechanisms for feedback suppression (of, e.g., ARQ NACKs), for increasing node density they are however affected by the so-called feedback implosion problem [5], i.e., control messages in this case collide with high probability thus dramatically impacting the performance. Also, a technique called *pipelining* [1] is used to push concurrency in the data transmission over multihop networks. Specifically, pipelining allows a node that correctly receives a data block from a neighboring sensor to immediately start its dissemination to the next hop. This increases the degree of parallelism in the data transmission phase and effectively decreases the programming time.

This paper presents SYNAPSE++, a reprogramming system built on SYNAPSE [6] that efficiently copes with the above requirements through the use of rateless fountain codes (FCs). These allow for high performance in dense as well as noisy environments and substantially mitigate the feedback implosion problem. The original contributions of this paper are:

- We design a new dissemination protocol consisting of an original pipelining strategy, coupled with a novel and distributed channel access mechanism, called soft TDMA.
- We improved the FC implementation of SYNAPSE by a joint design with the forwarding mechanism so as to maximize the number of errors that are corrected through overhearing, thus limiting the number of explicit retransmissions.

1536-1233/10/$26.00 © 2010 IEEE Published by the IEEE CS, CASS, ComSoc, IES, & SPS

- SYNAPSE++ features advanced boot loader and memory management modules, which allow the dissemination of binary images written in any operating system and make application and reprogramming software completely independent in terms of memory and variables.
- We provide an experimental evaluation of SYNAPSE++ in a real multihop deployment with 42 sensors and average path length of 8 hops.

Differently from previous reprogramming schemes, e.g., [1], [2], [3], [4], SYNAPSE++ is not embedded in the running application but it is rather a stand-alone program. We designed the boot loader so that at runtime it is possible to switch between SYNAPSE++ and any of the disseminated applications that, in turn, do not share RAM memory with SYNAPSE++ . This alleviates memory requirements and makes SYNAPSE++ suitable for the dissemination of third party proprietary software.

The rest of the paper is organized as follows: Section 2 surveys related work. Section 3 presents the design of the FC that we used in our framework and characterizes its performance. Section 4 describes the architecture of SYNAPSE++ along with the involved networking protocols. Section 5 presents our joint optimization of pipelining and FCs. Section 6 shows experimental results, where the performance of our dissemination system is compared with that of Deluge. Finally, in Section 7, we conclude the paper by discussing the tuning of SYNAPSE++ that we performed at the T. J. Watson IBM research center, where we used it as a support tool for a project on WSNs.

2 Related Work

In this section, we review a selection of reprogramming/data dissemination approaches for WSNs by subdividing them into five categories. A complementary taxonomy can be found in [7].

2.1 Earlier Approaches

XNP [8] is the first network reprogramming protocol for WSNs, it works over single-hop networks and does not support incremental updating of the program image. The Multihop over the air protocol (MOAP) [9] extended its functionalities to multihop networks and enhanced its data recovery phase through the usage of window and NACK-based ARQ (these features are all used by the most recent protocols). However, MOAP disseminates data in a hop-by-hop fashion, i.e., a node has to receive the whole program before it can start disseminating it over the next hop, and this may be inefficient in large multihop networks.

2.2 Backbone-Based Approaches

A second class of solutions is based on the construction of a connected dominating set (CDS), which is later used for a two-phase dissemination of the data. Sprinkler [10] is the first protocol that used this approach. It assumes location awareness at each node, which is used to construct a CDS. Objects metadata are disseminated through a packet-level pipelining scheme and a TDMA schedule is exploited for the transmissions among CDS nodes. Core nodes forward newly received data packets and piggyback the negative acknowledgment (NACK) for the lost data packets.

GARUDA [11] is a further dissemination protocol based on CDS. It uses a distributed and lightweight algorithm to approximate the CDS. It then exploits a two-phase error recovery that prioritizes the nodes in the CDS (core nodes) and then corrects the errors at the leaf nodes. Availability bitmaps and a modified ARQ policy are used for error recovery. A similar design is exploited by CORD [12], where the dissemination is also split into the above two phases. In addition, CORD further enhances the energy efficiency through the use of coordinated sleep schedules at each node.

2.3 Contention-Based Approaches

A third approach consists of disseminating the data by letting nodes randomly compete for the channel. This solution does not need the construction of a CDS and is more suitable for more dynamic topologies. Deluge [1] is the first dissemination system falling in this category. It propagates program images over multihop networks through an epidemic routing approach. Data transmission and NACK-based ARQ are jointly implemented through a three-way handshake mechanism based on advertisement (ADV), request (REQ) and actual code (CODE) transfer. Deluge copes with the memory constraints of sensor nodes by splitting the program image into pages. The image is then transmitted page-by-page exploiting broadcast transmissions and pipelining. In addition, further features such as ADV suppression and randomization of the transmission of ADVs within predetermined time windows are implemented to reduce the congestion during the dissemination phase. Most of these techniques (especially pipelining) have then been exploited by all subsequent protocols.

A further protocol, MNP [2], additionally implements special algorithms to reduce the problems due to collisions and hidden terminals. This is achieved through a distributed priority assignment so that, within a neighborhood, there is at most one sender transmitting the program image at any given time. The sender election is greedy and distributed: the senders with a higher number of potential receivers are assigned higher priority, and sleeping modes are used to reduce energy consumption.

Freshet [3] builds on Deluge by aggressively optimizing the energy consumption during reprogramming. In an initial phase, some metadata about the image to be transferred and the topology (in terms of number of hops from the front wave where the code is currently being transmitted) are disseminated to sensor nodes. Using this information, nodes estimate when the image will actually reach their vicinity and enter a sleeping period accordingly.

According to their current TinyOS implementation, Deluge, MNP, and Freshet all disseminate the image of the programming protocol together with that of the program to be transferred. However, this considerably inflates the amount of data to be disseminated. Stream [4] fixes this problem by preinstalling in each sensor node the reprogramming application. This is done through the segmentation of the FLASH into multiple partitions so that the reprogramming protocol and the program to be transferred are stored in different image areas. Hence, at dissemination time Stream transmits over the channel the minimal support (about one page) needed for the activation of the reprogramming image together with the actual program image. A last

scheme, Typhoon [13], is also based on the ADV-REQ-CODE paradigm, but it transmits the image using multiple frequency bands so as to further push the concurrency of the transmission (spatial reuse).

2.4 Dissemination Based on Rateless Codes

Recently, researchers started to use rateless codes, as we do in this paper. Rateless-Deluge [14] enhances [1] through a HARQ technique based on rateless codes on Galois fields of size 2^q with $q = 8$ ($GF(2^8)$). AdapCode [15] is a further scheme exploiting network coding on $GF(2^5)$. In AdapCode, packets are coded at every node through linear combinations with coefficients picked randomly in the Galois field. Also, the coding aggressiveness is adaptively changed according to link qualities and number of neighbors. SYNAPSE [6] implements a HARQ similar to that in [14] using $GF(2)$, which is however optimized for hop-by-hop data dissemination.

The protocol that we present in this paper, SYNAPSE++, builds on SYNAPSE by adding full support for pipelining through a joint design of MAC and fountain codes. Specifically, it adopts the above ADV-REQ-CODE paradigm, it randomizes the transmission of ADVs to avoid collisions, and it exploits pipelining and implements the optimization of [4]. It also features new elements such as the extension of Deluge's FLASH memory partition management as well as a novel hybrid ARQ transmission and error recovery mechanism based on rateless fountain codes [16]. Differently from previous work using rateless codes, our scheme uses random codes over $GF(2)$, i.e., encoding and decoding only need to perform bit-wise XORs among packets. We note that while working with fields of larger size is more effective in terms of transmission overhead and recovery capabilities, it is less efficient in terms of decoding time and computational resources on sensor nodes, as operations in this case are rather slow as compared to the standard XORs required in $GF(2)$.

In SYNAPSE++, we compensated for the lower performance of $GF(2)$ through an optimization (via genetic algorithms) of the distribution used at the encoder so that our FC still provides overhead and recovery capability performance close to that achievable for larger fields. As an example, SYNAPSE++'s optimized decoding of 32 packets takes about 460 ms as compared to delays of the order of 3 s for $GF(2^8)$ [14] (see Section 3.3 for further details). Also, SYNAPSE++ implements an original pipelining strategy with transmission priorities based on a loose TDMA synchronization within nodes in the same neighborhood. This enables the separation of ADV, REQ, and DATA phases so as to minimize collisions, facilitate pipelining, and improve the effectiveness of FCs. Finally, SYNAPSE++'s pipelining scheme and FCs are jointly designed and optimized. To the best of our knowledge, SYNAPSE++ is the first dissemination system to adopt this joint design.

2.5 Further Design Choices

In this last category, we describe a few more techniques that are complementary to the actual transmission process and that can be used to further improve the efficiency of the overall reprogramming system. Dunkels et al. [17] present dynamic linker and loaders, using the standard ELF format and show that dynamic linking is effective for reprogramming resource constrained sensor nodes. This method is used by recent operating systems for WSNs such as Contiki [18]. Along the same line, Flexcup [19] optimizes the way the software running on the nodes is linked together. It generates metadata, describing the compiled components. During the code dissemination, this metadata is used to install new components into the running application, re-link function calls and perform address binding of data objects. This allows to install parts of applications without having to disseminate the entire program image. Lightweight data compression schemes for WSNs with small memory footprint are presented in [20]. The idea is to compress the program image, disseminate it through any of the above protocols, and then obtain the original program through decompression at the receivers. Appropriate compression schemes can save energy and reduce dissemination time with respect to sending the uncompressed application. Tsiftes et al. [20] also provide important hints on the trade-off between immediately writing received data to FLASH and buffering it in RAM, which are important considerations for SYNAPSE++'s design.

3 Fountain-Based Encoding

The description of our fountain code is split into three sections. In Section 3.1, we discuss the main characteristics of fountain codes, why we use them in our framework and their main differences from other encoding methods. In Section 3.2, we specify the fountain code that we designed for SYNAPSE++, discussing the optimizations carried out at both transmitter (encoder) and receiver (decoder) sides. Finally, in Section 3.3, we illustrate important implementation details.

3.1 Introduction to Fountain Codes

Digital fountain codes were presented by Luby in [21]. They are near optimal rateless codes designed for binary "erasure channels." A binary erasure channel is such that each transmitted packet is either correctly received without errors or entirely lost with probability p, which identifies the *cannel erasure probability* [16], [22].

A fountain code is conceptually very simple. For illustration, consider Fig. 1 and let $x_1, \ldots, x_5$ be the five source packets that need to be transmitted between a sender and multiple receivers and consider a zero erasure probability, $p = 0$. Instead of transmitting the source packets, we apply an FC so that each transmitted packet is obtained as a linear combination of a subset of the input packets $x_1, \ldots, x_5$. In the example of Fig. 1, seven encoded packets y_i, $i = 1, \ldots, 7$ are transmitted, where, e.g., y_1 is obtained as the bit-wise XOR of x_1 and x_2. The number of x_i's to be summed together is picked according to a "degree distribution," as we explain shortly, whose design is key to obtaining good performance. Moreover, y_i and y_j with $i \neq j$ are statistically independent, i.e., the input packets picked for y_i do not depend on those picked for y_j.

The relationship between x_i and y_i is captured by the matrix transformation on the left of (1), which can be written as $\mathbf{y} = \mathbf{G}\mathbf{x}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors containing the x_i and y_i and $\mathbf{G}$ is the transformation matrix.

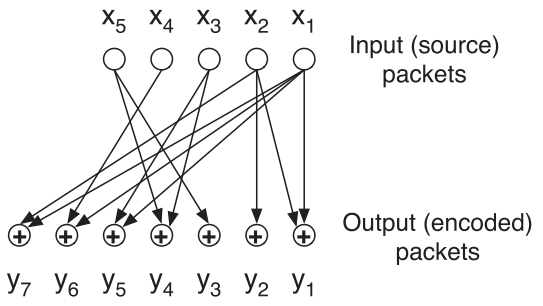

Fig. 1. Fountain Codes: encoding example.

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

$$\underset{\text{Gaussian elimination}}{\Rightarrow} \begin{pmatrix} y_1 \oplus y_2 \\ y_2 \\ y_3 \oplus y_4 \\ y_1 \oplus y_2 \oplus y_6 \\ y_3 \\ \text{irrelevant} \\ \text{irrelevant} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}. \quad (1)$$

Each encoded packet is associated with a so called *encoding vector* that, for y_i, is nothing but the ith row of $\mathbf{G}$. Encoding vectors are sent along with encoded packets and used at the receivers to obtain $\mathbf{x} = \mathbf{G}^{-1}\mathbf{y}$. Hence, all receivers can invert this transformation, e.g., through Gaussian elimination, and retrieve $x_1, \ldots, x_5$ through bitwise sums of some encoded packets, as shown in (1). For $p > 0$, the received matrix $\mathbf{G}$ might not be invertible (i.e., its rank may be less than 5) at some receivers (as the channel might now "erase" some of the transmitted packets) and in this case the x_i cannot be retrieved. Nevertheless, with fountain codes it is possible to retrieve the five original packets at all nodes through the transmission of additional encoded packets, until $\mathbf{G}$ is full rank at all receivers. This is why FCs are said to be "rateless," i.e., the rate of the code can always be extended on the fly depending on the number of packets lost.

Common methods for reliably transmitting packets over erasure channels are ARQ protocols where receivers send back to the transmitter status reports to identify missing packets. The transmitter, in turn, decodes incoming reports and retransmits what is lost. ARQ works regardless of the erasure probability p but often requires a large amount of feedback. In addition, the forward channel (transmitter $\rightarrow$ receivers) performance (e.g., delay and throughput efficiency) is heavily impacted even for a small number of receivers and low error rates [23]. As a solution, in the literature several HARQ schemes have been proposed, see, e.g., [5], [23]. HARQ scales considerably better than ARQ as a single redundancy packet can recover different losses at multiple receivers. We advocate the use of fountain codes-based HARQ schemes for programming WSNs as these retain the good performance of previous HARQ schemes [5], [23] while presenting additional advantages:

- Due to the rateless nature of these codes, we do not need to know in advance the error probability p. This simplifies implementation and increases efficiency. In fact, the actual amount of redundancy to use within our dissemination protocol can be decided on the fly, while retaining the good error recovery performance of traditional HARQ schemes that use packet-based Reed Solomon [5], [23] or Tornado [24] codes.
- Packets are encoded using arithmetic on the Galois field $GF(2)$, i.e., by means of bitwise XOR operations. This substantially speeds up the execution time with respect to traditional packet-based Reed Solomon codes [5], [23], which use more complex operations among polynomial coefficients in $GF(2^q)$, with $q > 2$. In fact, fast operations over $GF(2^q)$ require the use of lookup tables, which is substantially slower than XORing symbols. This is a tremendous advantage for resource constrained sensor devices. We also observe that, while Tornado codes [24] also perform encoding in $GF(2)$, they are not rateless.

3.1.1 Encoding Procedure

The key ingredient is the *degree distribution* $\rho(d)$, which is a probability distribution, determining the number of input packets to combine to form any given encoded packet y_i. The input file is subdivided into a number of, say, K packets, and the following operations are executed:

- Pick a degree d_i, $1 \leq d_i \leq K$ from the distribution $\rho(d)$, whose characteristics depend on the number of original packets K in the input file, as well as on the targeted performance (e.g., in terms of coding complexity and overhead, see Section 3.2).
- Randomly and uniformly pick d_i packets among the K given as input. The encoded packet y_i is obtained through the bitwise, modulo 2 sum of these d_i packets, i.e., by successively XORing them. d_i is the *degree* of the encoded packet so obtained, while the information about which d_i packets were XORed together forms the corresponding *encoding vector* and needs to be known at the decoder. Continue from the previous step until the desired number of packets is encoded.

Note that, due to the above procedure, all encoded packets are equally representative of the whole input file, as they are independently generated using the same distribution. Hence, it is not important which packets are lost during transmission, but rather what matters is how many packets are correctly received. Moreover, the goodness of the encoding process is totally captured by the adopted *degree distribution*, whose optimization is thus crucial to obtain good performance. This optimization is the subject of the following Section 3.2.

3.1.2 Decoding Procedure

Decoding can be done by inverting the decoding matrix $\mathbf{G}$, which is formed by the received encoding vectors, i.e.,

solving for $\mathbf{x}$ the system $\mathbf{y} = \mathbf{G}\mathbf{x}$, where $\mathbf{y}$ is the vector containing the received encoded packets, whereas $\mathbf{x}$ contains the K original packets to be retrieved (see (1)). Of course, a necessary condition for this inversion is that $\mathbf{G}$ has full rank K. In general, this may require the collection at the receiver of $N > K$ encoded packets as, due to the random encoding method of FCs, not all encoding vectors are guaranteed to be linearly independent. The actual number of packets N that need to be received to obtain a full rank matrix depends on the selected distribution $\rho(d)$. We define the overhead O as the extra redundancy needed for the recovery of the K original packets, i.e., $O = N - K$.

In practice, for large K (usually larger than 1,000) there are encoding distributions providing a small overhead as well as very efficient decoding procedures [21] (these are based on message passing and heuristically solve the linear system $\mathbf{y} = \mathbf{G}\mathbf{x}$). Our focus in this paper is however different as K in our setting is typically small. Here, K is the number of packets in a *transport block*, that is just a portion of the whole program image. We recall that, due to the inherent memory limitations of sensor devices, we cannot work with large K values.[1] Hence, the suboptimal decoding in [21] is not a useful option in our case due to its poor performance in terms of overhead for small K. On the other hand, we note that optimal decoding (in terms of decoding overhead) amounts to inverting the decoding matrix $\mathbf{G}$, which can be done through, e.g., Gaussian elimination and back-substitution [22]. For large K, this method is not efficient, as its complexity grows as $\mathcal{O}(K^3)$. However, in our case this complexity is acceptable due to the small values of K (e.g., $K = 32$). Hence, we decided to implement an optimal decoder, according to an efficient Gaussian elimination routine. This, together with the optimization of the degree distribution at the encoder, led us to small decoding overhead at the cost of a reasonable complexity.

3.2 Optimization of the Degree Distribution $\rho(d)$

Properly designed fountain codes should have N close to K. Some overhead is unavoidable and depends on the adopted degree distribution $\rho(d)$. In this section, we present an original and very effective algorithm for the optimization of $\rho(d)$ according to given performance objectives. The optimized degree distributions that we obtain in this section are used within SYNAPSE++'s error recovery scheme. We also stress that *they are general as they do not depend on the particular program image to transmit*: this is why $\rho(d)$ is only used to decide how many source packets are to be combined, regardless of their actual content.

We optimize our FCs for transmission over error-free channels. In fact, for full recovery at the receiver(s), it is sufficient to receive K independent packets so that $\mathbf{G}$ can be inverted. This implies the reception of $N \geq K$ packets as not all packets we generate through $\rho(d)$ are linearly independent. However, an error probability $p > 0$ does not alter the decoding operations at the receiver side as K independent packets are still needed. Hence, as packets are generated independently of each other, losses will preserve all the properties of the distribution designed for $p = 0$. In practice, a good distribution for error-free channels will preserve its good performance over error-prone links [22].

Before describing our optimization algorithm, we introduce a few definitions. A *sample* of the algorithm involves the generation of encoded packets until these allow full recovery at the decoder. An *iteration* of the algorithm is composed of a fixed number of samples, M. To optimize the degree distribution, we adopt an iterative approach: we start from an initial distribution, we generate samples and, for each of them, we calculate a cost, which is subsequently used to refine the distribution itself. The procedure is terminated when a *stopping condition*, which is defined below and depends on the latest distribution obtained, is verified. A new iteration is started otherwise. In the following, we define some parameters:

- K: number of packets in the input file.
- $p_j, j = 1, 2, \ldots, K$: point probabilities defining the degree distribution $\rho(d)$.
- M: number of samples generated during each iteration.
- $C(i), i = 1, 2, \ldots, M$: cost associated with the ith sample of the current iteration.
- $N(i), i = 1, 2, \ldots, M$: number of encoded packets needed to retrieve the original K packets for sample i.
- $n_j(i), i = 1, 2, \ldots, M$; $j = 1, \ldots, K$: degree j packets generated within the ith sample, $\sum_{j=1}^{K} n_j(i) = N(i)$.

We use ideas from the theory of genetic algorithms to iteratively obtain an optimized degree distribution. We start by generating a population of M samples and evaluating for each sample i its cost $C(i)$. $C(i)$ may, for example, be a function of the overhead (defined as $O(i) = N(i) - K$) and/or of the number of elementary operations (XORs) required for decoding. Once we have the costs for all samples $1, 2, \ldots, M$, we select the most *promising* samples as follows: We compute the α-percentile, C_α, of the observed costs $C(1), C(2), \ldots, C(M)$ and pick all samples k having cost $C(k) \leq C_\alpha$. These samples are subsequently used to refine the degree distribution $\rho(d)$. The refined distribution *survives* to the next iteration. Let $\mathcal{S}$ be the set containing the selected samples: $\mathcal{S} = \{k : C(k) \leq C_\alpha\}$ and let p_j be the point probabilities associated with the current distribution. The new distribution is obtained as:

$$p_j^{new} = \frac{\sum_{k \in \mathcal{S}} \frac{n_j(k)}{N(k)}}{|\mathcal{S}|}, \quad j = 1, 2, \ldots, K, \tag{2}$$

where $|\mathcal{S}|$ is the cardinality of set $\mathcal{S}$. Also, it is easy to verify that $\sum_{j=1}^{K} p_j^{new} = 1$. In fact,

$$\begin{aligned} \sum_{j=1}^{K} p_j^{new} &= \frac{1}{|\mathcal{S}|} \sum_{j=1}^{K} \left(\sum_{k \in \mathcal{S}} \frac{n_j(k)}{N(k)} \right) = \frac{1}{|\mathcal{S}|} \sum_{k \in \mathcal{S}} \sum_{j=1}^{K} \left(\frac{n_j(k)}{N(k)} \right) \\ &= \frac{1}{|\mathcal{S}|} \sum_{k \in \mathcal{S}} \frac{N(k)}{N(k)} = 1. \end{aligned} \tag{3}$$

Since this new distribution is obtained from the smallest cost samples, it is reasonable to suppose that adopting p_j^{new} for the generation of new samples, i.e., at the next iteration of the algorithm, will result in outcomes with smaller cost. These are in turn used to generate a new distribution, and

1. More details on how the application is split for transmission over the network are given in Section 4.3.

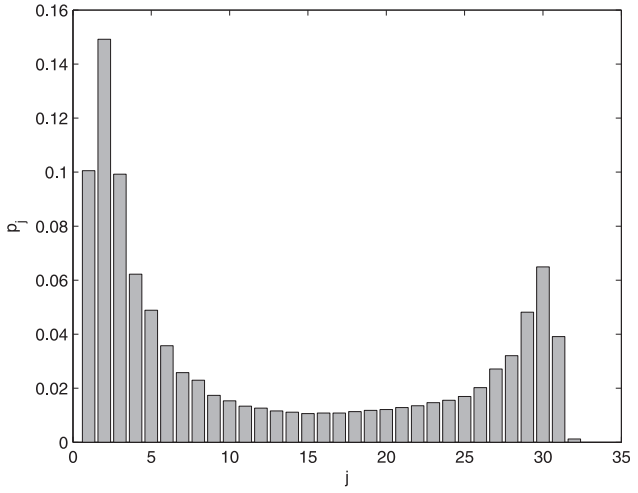

Fig. 2. Selected optimal encoding distribution $\rho(d)$ for $K = 32$.

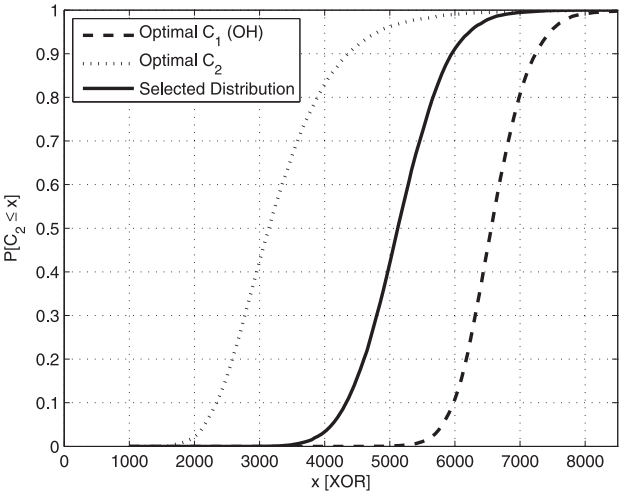

Fig. 3. Cdf of the computational cost at the decoder for different $\rho(d)$. The cost is measured as the number of XORs between 16-bit words.

this procedure is iterated until a certain stopping condition is verified. In our algorithm, the stopping condition is defined in terms of the expected value of the cost during the last two iterations. In particular, the optimization process is continued if and only if the mean cost obtained in the current iteration is strictly lower than that obtained previously. We tested our algorithm, comparing its performance against that of state-of-the-art optimization schemes [25]. Results demonstrating that our genetic approach substantially outperforms [25] can be found in [6].

As discussed in Section 3.1, the message passing (LT) decoder [21] is not suitable for our settings, i.e., for small K values. On the other hand, in our case an optimal decoder, based on Gaussian elimination, can be used at a reasonable computational cost. We thus applied our optimization algorithm to a Gaussian elimination decoder to obtain distributions having *low decoding cost* as well as *low overhead*. For this purpose, we used two cost functions: $C_1(i) = N(i)$, accounting for the number of packets needed to decode, and $C_2(i)$, which is defined as the number of XORs performed to recover the original K packets. These two cost functions were used to define a new set of *useful* samples $\mathcal{S}' = \{k : C_1(k) \leq C_{\alpha_1} \bigwedge C_2(k) \leq C_{\alpha_2}\}$, to be used in (1), that in this case was obtained considering two different values for the α-percentiles for C_1 and C_2.

A new set of optimal encoding distributions $\rho(d)$ was thus found for $K \in \{32, 48, 64, 128\}$ through the above genetic algorithm with the two costs C_1 and C_2, where $p_j = 1/K$, $j = 1, 2, \ldots, K$ was used to initialize the optimization process. By tuning the two parameters α_1 and α_2, we selected the following distributions $\rho(d)$: D1) $\alpha_1 = 0.05, \alpha_2 = 1$: having minimum overhead ($\mathbb{E}[C_1] = 33.65$ packets, $\mathbb{E}[C_2] = 6{,}586$ XORs), D2) $\alpha_1 = 1, \alpha_2 = 0.05$: having minimum decoding cost ($\mathbb{E}[C_1] = 50.7$ packets, $\mathbb{E}[C_2] = 3{,}249$ XORs), D3) $\alpha_1 = 0.05, \alpha_2 = 0.075$: achieving a suitable trade-off ($\mathbb{E}[C_1] = 34.26$ packets and $\mathbb{E}[C_2] = 5142$ XORs).

D1 and D2 obtain unsatisfactory performance in terms of cost and overhead, respectively. D3 was instead selected as the final $\rho(d)$ to use in SYNAPSE++ as, with respect to D1, its overhead is just slightly larger and its decoding cost is reduced by about 20 percent. D3 is shown in Fig. 2, whereas in Fig. 3 we show the complementary distribution function (cdf) of the decoding cost for D1, D2 and D3. We stress that D3 does not depend on the input data content, and that the genetic algorithm is only run offline to find this distribution and not during SYNAPSE++'s data dissemination process. With larger K, e.g., $K = 128$ a cost reduction of up to 40 percent can be achieved with respect to D1, whilst maintaining almost the same overhead. This K, however, hardly fits our memory requirements. Further details about the various trade-offs are not given here due to space constraints and can be found in [6].

3.3 Implementation Details

First of all, encoding vectors are not transmitted along with encoded packets. We instead use the same random number generator at both transmitter and receivers and associate random seeds with encoded packets. In our implementation, each packet carries a 16 bit field containing the seed that was used to generate it. Thus, at the receiver side, seeds are used to synchronize the pseudorandom generator with that used at the encoder (transmitter) and reproduce the encoding vector for any received packet. This improves robustness and facilitates error correction through overhearing (see Section 4.3).

In SYNAPSE++, we adopt a generator based on Linear Feedback Shift Registers (LFSR) [26], which works with registers of 16 bits. This method, which is optimized for the TI MSP430 microcontroller of our TmoteSky sensor nodes, is very fast. Decoding a block of $K = 32$ packets (800 bytes) with LFSR takes 462 ms. A drawback of LFSR is that a few random seeds exist which provide unsatisfactory performance. There are, however, a large number of seeds for which LSFR performs properly.

In Fig. 4, we plot the time taken for the TI MSP430 to successfully decode a transport block of K data packets (of 25 bytes each) using $GF(2^8)$ (as used in, e.g., [14]) and compare it to that of SYNAPSE++'s optimized decoder. The observed improvement of a factor no less than 6.5 is due to 1) performing operations over $GF(2)$ (bitwise XORs) and 2) adopting our optimized degree distributions, which make the decoding matrix $\mathbf{G}$ sparse. On the other hand, for $GF(2^8)$ decoding K original packets requires the reception of exactly K packets, whereas $GF(2)$ reaches the same goal

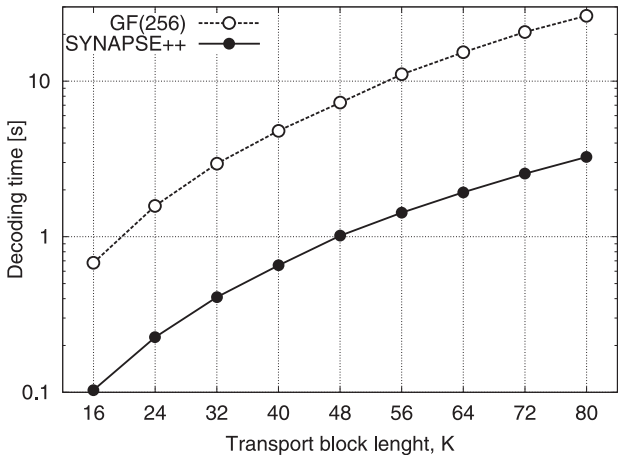

Fig. 4. Decoding time: comparison between Gaussian elimination with $GF(2^8)$ and SYNAPSE++.

with a larger number of packets, e.g., 36 (this overhead is also taken into account in Fig. 4). Hence, using SYNAPSE++ we have an additional time overhead of 13 percent (transmission of four additional packets) that has to be compared to a time dilation of 550 percent that comes from using $GF(2^8)$.

Finally, we observe that in addition to picking good seeds in terms of their decoding times, we also had to optimize them so that different nodes transmitting the same original data will produce, with high probability, linearly independent encoded packets. This fact is exploited in SYNAPSE++'s pipelining strategy, which is presented in Section 4.3. The optimization of the fountain code when used with pipelining techniques is instead discussed in Section 5.

4 SYNAPSE++'s Design

We structured the discussion of SYNAPSE++'s architecture and algorithms as follows: Section 4.1 describes SYNAPSE++'s architecture, Section 4.2 introduces the Boot Loader, which we realized to allow the management of the FLASH (formatting, partitioning, etc.) and to load new programs. Section 4.3 presents the fountain-based dissemination protocol. The architecture was designed and implemented for the TmoteSky sensor nodes (the same hardware is used by TelosB motes) and programmed in TinyOS 2 [27].

4.1 General Architecture

It is important to note that SYNAPSE++ is not a service embedded in the distributed code, but instead it is a standalone application. This is a design choice that offers two main advantages. First, distributed applications do not require additional code to embed the service, which would make them larger. Second, the applications that are disseminated using SYNAPSE++ are completely independent of the dissemination service itself and, as such, they are not constrained to be developed in a specific language or operating system. All applications are seen as plain binary code, which allows the distribution, for example, of closed-source programs. The boot loader, which is always present in the bootstrap sector of the microcontroller, handles the switch between the dissemination operation and the execution of the many applications that can be distributed and stored in the nodes.

Due to this complete separation, when a node switches to SYNAPSE++ the whole RAM is available. An application is usually larger than the amount of RAM present on the nodes, thus it has to be split into pages before being distributed. The pages are decoded and stored in the external FLASH. Since the radio chip and the external memory often share the same bus they cannot operate at the same time. This means that while storing a page on the external memory a node is unaware of any incoming messages. To limit the number of times this occurs, the current implementation of SYNAPSE++ keeps a page size of $K = 32$ packets, which requires an amount of memory of about 1.5 kB (for a total overall occupation of about 4 kB). This is supported by all existing sensor platforms, to the best of our knowledge, but could be easily modified if needed. More details on the switching between application and SYNAPSE++ are given in Section 4.2.

Next, we describe the architecture of SYNAPSE++, detailing its functional blocks.

4.1.1 Structure of the Software

The software was developed in a modular and portable way to facilitate its extension/modification. There are three macroblocks: the Boot Loader (`BL`), the data dissemination manager (`DDM`) and the external memory (`EM`). The communication between `BL` and `DDM` is possible thanks to the information memory (IM), a portion of the internal memory that is preserved even after a device reset (non-volatile memory). Further modules are contained in the `DDM`, namely, the boot loader communication (`BLC`), the data dissemination logic (`DDL`), the partition manager (`PM`), the `Radio` and the `Codec`. Next, we detail their functionalities as well as their interactions.

4.1.2 Boot Loader

Our sensor nodes feature a TI MSP430 micro-controller with direct access to an internal memory of 48 kB; additional storage is provided by 1,024 kB of FLASH memory (External Memory, `EM`). The `BL` is loaded at boot time, and handles read and write operations between these memories. Also, it loads applications residing on the `EM` on demand; these applications, as we explain below, can be written in any programming language. Due to its importance, we present the Boot Loader in greater detail in Section 4.2.

4.1.3 Data Dissemination Manager

This is the core of our data dissemination system. It comprises the Data Dissemination Logic, the `Radio`, the `Codec`, the `BLC`, and the `PM`. These modules are described in what follows.

4.1.4 Data Dissemination Logic

It orchestrates the `Radio` and `Codec` modules in order to send, encode and decode data and control packets. This module makes local forwarding decisions, i.e., it decides, upon the reception of a new *transport block* (TB), whether the receiving node should act as a forwarder for this transport block, handles the transmission of transport blocks (through the `Radio` module) and implements the error control policy (for data recovery, including joint coding/retransmissions and pipelining). Also, it exploits a dedicated soft-TDMA

scheduling to synchronize neighboring nodes and improve the performance of the pipelining scheme.

4.1.5 Radio

This module provides the `Codec` with the current set of original packets (a block of K packets in our case), which are used by the `Codec` to obtain encoded packets. These are then passed back to the `Radio` module for their actual transmission over the channel. For improved efficiency, we synchronized the coding and sending procedures so as to enable back-to-back transmission of data packets. In detail, soon after passing a packet to the radio transceiver, the `Radio` module triggers the `Codec` for the generation of the next encoded packet.

4.1.6 Codec

The codec implements the fountain code coding routines, specifically designed to address the constraints of wireless sensor nodes. The fountain code used by this module has been further optimized, with respect to [6], for its usage in conjunction with pipelining techniques. These aspects are treated in deeper detail in Sections 3 (fountain code design) and 5 (pipelining).

4.1.7 Boot Loader Communication

This is a TinyOS module allowing the communication between TinyOS applications (in our case the Data Dissemination Manager) and the Boot Loader. It is intended to provide an abstraction to the actual hardware. The module allows the application to access the information segment and to issue *reboot* and *load* commands.

4.1.8 Partition Manager

Partition manager provides functionalities for reading, writing, and creating memory partitions in the external FLASH. The `PM` component maintains a persistent partition table, stored at the beginning of the FLASH memory. This allows the dynamic allocation of new partitions when new applications are received. Using a partitioning system as an abstraction from the hardware makes the system more flexible and allows an easy migration to other platforms.

4.2 Boot Loader

The Boot Loader manages many operations on the external storage (i.e., formatting, saving new applications and exchanging data between the internal and the external memories). Before issuing the *reboot* command, an application writes in the information memory, which is nonvolatile, information about the operation that has to be executed at the next reboot by the Boot Loader. The main commands are: *Format* the `EM`, *Store* and *Load* applications. The `EM` is a Write Once Read Many (WORM) device. Thus, we wrote our dynamic partitioning system to work with WORM memories without knowing in advance the number and the size of the partitions.

Each application that we store in the `EM` is uniquely identified by an *application ID*.[2] In the first partition of the external memory, we maintain a *partition table* which relates application IDs to the memory location where the corresponding application is saved. The application ID is subsequently used by the boot loader to retrieve the application from the external FLASH, copy it to the internal memory and load it. In detail, each entry of the partition table contains the following pieces of information: the application ID, the address, and the size of the application in the `EM`, and a flag that indicates whether the application passed the CRC check, i.e., whether it is complete and correctly received (this flag is used to prevent the system from loading incomplete or corrupted images).

4.3 Data Dissemination Protocol

Next, we present the data dissemination and error recovery algorithms of SYNAPSE++. Our aim is to disseminate a program image to all nodes of a WSN. Due to the inherent memory constraints of sensor nodes, an efficient dissemination requires splitting files into B transport blocks (TBs) of K packets each, so that they can be processed in the available RAM. Transport blocks are then encoded through our fountain code into $K' = K + \delta$ packets each (δ is the number of redundant packets) before being transmitted. As its predecessors [1], [2], [4], SYNAPSE++ uses an ADV/REQ/DATA transmission paradigm, which is coupled with an original pipelining and coding scheme. Below, we describe SYNAPSE++'s protocol elements.

4.3.1 ADV/REQ Handshaking

Any given node n maintains a bit-mask $b(n)$ indicating the TBs that it correctly received so far. As done in [1], [2], [4], either periodically or whenever a new TB is received, any node n advertises its status $b(n)$ by sending an ADV. This informs its neighbors about the TBs that this node can provide and allows out-of-order delivery of transport blocks within the network (this feature dramatically improves the performance with respect to sending blocks in order). Interested neighbors respond with an REQ message including the smallest identifier among the blocks they need. In case multiple REQs are received, n will serve the one requesting the highest block (this is done to promote advancement).

In addition to $b(n)$, every ADV also carries an indication of the hop count of the sending node as well as the identifier and the total size of the program image. Each node keeps sending ADVs as in [1] (doubling the inter-ADV time whenever a node does not receive a corresponding REQ) until a stopping command is received from the sink node.

4.3.2 Fountain Codes

First of all, fountain code encoding is applied to each TB. That is, given the K original packets of a given TB, FC produces a slightly larger number of packets K', where the overhead $\delta = 4$ is picked to have a successful recovery with probability greater than 0.8 at the first transmission of the TB (i.e., of these K' packets) considering a typical packet erasure probability $p = 0.1$. In detail, at the receiver the K original packets are successfully decoded with probability $P_s(\delta) = \sum_{\xi=0}^{K+\delta} \Psi(K+\delta-\xi) P_\xi(K+\delta)$, where

$$P_\xi(K+\delta) = \binom{K+\delta}{\xi} p^\xi (1-p)^{K+\delta-\xi}$$

2. This identifier could be obtained through, e.g., a hash function, calculated as a function of the application object's code. This generation is performed by the compiling system, which usually resides in a PC having the necessary computational power.

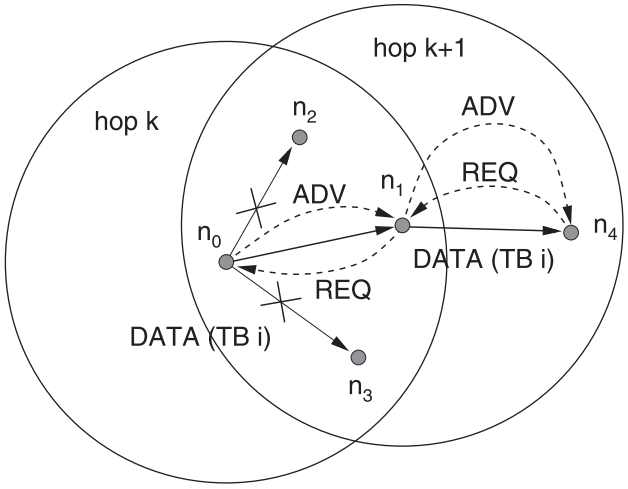

Fig. 5. ADV/REQ/DATA handshaking example.

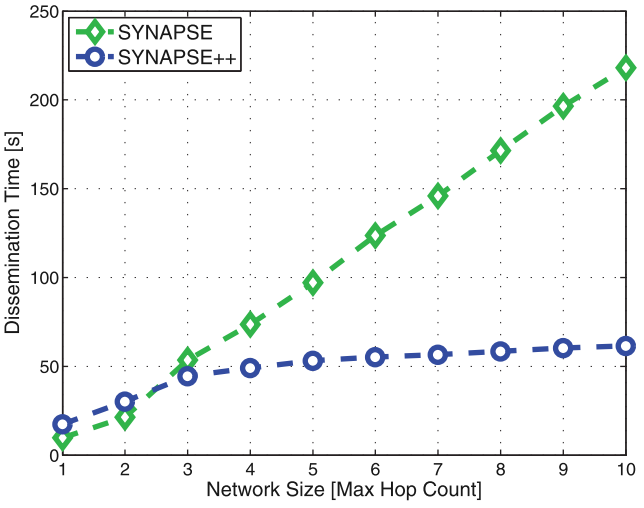

Fig. 6. Simulation: dissemination time of SYNAPSE and SYNAPSE++.

is the probability of having ξ erasures over $K+\delta$ transmissions. $\Psi(x)$ returns the probability that the correct reception of x encoded packets leads to a full rank $\mathbf{G}$. An excellent approximation for $\Psi(\cdot)$ can be found through numerical simulation, as it only depends on the selected $\rho(d)$. In our protocol, each time a node n has a TB to send, it picks a suitable random seed s and obtains the K' encoded packets for this TB as explained in Section 3. To achieve good performance, the seeds have to be carefully selected as explained in Section 5.

4.3.3 Pipelining

This technique is used to allow concurrent transmission of the same TBs in different portions of the network. While SYNAPSE++ also leverages this transmission paradigm, it features a novel approach as 1) packets are encoded through a suitable FC, 2) transmission turns among nodes are coordinated through an original pipelining scheme exploiting soft-TDMA schedules for improved efficiency, and 3) pipelining and FCs are coupled through the selection of proper seeds to encode and transmit data over subsequent hops. In this way, SYNAPSE++ jointly performs error correction and data forwarding.

Let us now refer to the example in Fig. 5: after a first ADV/REQ/DATA phase for TB i from node n_o, using seed s_o and broadcasting K' encoded packets, we have that node n_1 successfully decodes this block while nodes n_2 and n_3 fail. Our previous version of SYNAPSE, in this case, would have gone through a number of transmission rounds, sending (from node n_o) additional encoded packets (incremental redundancy) to recover from failures. Instead, SYNAPSE++ prioritizes the advancement of data with respect to local error recovery. Accordingly, n_1 occupies the channel and enters its own ADV/REQ/DATA phase for TB i: in detail, node n_1 sends an ADV for TB i, n_4 responds with an REQ and n_1 broadcasts K' new encoded packets for TB i using a different seed $s_1 \neq s_o$. Now, full recovery of TB i at any unsuccessful node (in this case n_2 and n_3) occurs if this node received at least K independent encoded packets out of the $2K'$ transmitted (including the first transmission from n_o and the second from n_1). We stress that in this way we can concurrently advance toward the next hop while recovering the losses at the unsuccessful nodes belonging to the last visited hop.[3] A similar overhearing technique is also used when packets are sent uncoded as in [1], [2]. However, in SYNAPSE++ any additional linearly independent packet increases the rank of the decoding matrix $\mathbf{G}$ and thus can be used to recover a partially received block, whereas according to plain ARQ schemes nodes need to receive the exact packets that have been lost. The policy adopted for the selection of the seeds is crucial to the success of this pipelining strategy as it should maximize the probability that, e.g., seeds s_o and s_1 will lead to linearly independent encoded packets. This is treated in greater detail in Section 5.

The effectiveness of the pipelining strategy can be seen from Fig. 6, where we compare the reprogramming time of SYNAPSE++ against that of SYNAPSE [6] (which exploits hop-by-hop transmissions). Each point in this figure was obtained with the event-driven simulation tool presented in [28], averaging the results over 100 different random topologies with size ranging from 1 to 10 hops. SYNAPSE exploits a hop-by-hop dissemination scheme, which is beneficial for small networks, i.e., on average up to 3 hops, as confirmed by the analysis in Section 5.2.4 of [1]. For increasing network size SYNAPSE++ takes advantage of its new dissemination method and performs significantly better.

4.3.4 Synchronization and Priority

In order to reduce the number of collisions and avoid idle times, which are typical of pure CSMA schemes, we opted for a loosely synchronized channel access scheme, which we call soft-TDMA (and that we use in substitution of the TinyOS MAC). According to this technique, ADV/REQ/DATA phases follow a specific time frame structure (hereafter referred to as *frame*), see Fig. 7. Specifically, the transmission is subdivided into three time intervals: the first, t_{ADV}, is dedicated to the transmission of ADVs, the second, t_{REQ}, is for the transmission of REQs and the last, t_{DATA}, is allotted to DATA transmission and decoding. ADV and REQ intervals are further subdivided into ℓ_{ADV} and ℓ_{REQ} access slots (of duration t_{Slot}), respectively.

3. When overhearing does not lead to TB recovery, local retransmissions are performed to recover the erroneous block. These have lower priority with respect to other forwarding operations so as to take advantage of the transmission of the same TB by different neighbors.

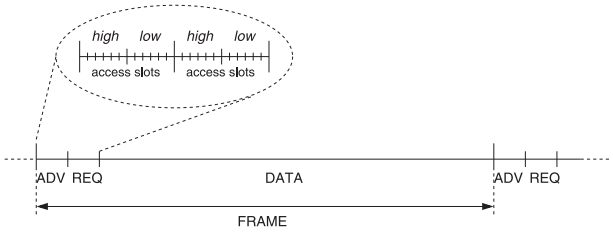

Fig. 7. ADV/REQ/DATA time *frame* structure.

The originator of the dissemination sets the reference time for the first *frame* within its first hop neighborhood. It does so by including in its ADV the lapse of time, Δt_{ADV}, to the next REQ interval. Access slots of ADV and REQ intervals are further subdivided into two priority levels. In particular, the first half of their slots are assigned high priority. Priorities are used to push newest data toward unexplored portions of the network; our aim here is to facilitate pipelining, thus quickly reaching a high degree of parallelism in the data transmission. Upon successful decoding of a new TB, a node sets its ADV priority to *high*. This priority level lasts for the next time *frame* and will be set back to *low* upon its completion.

Getting back to our example in Fig. 5, let us now consider node n_1 after the reception of the new TB i. For the sake of illustration, we assume that this TB is the highest correctly received by n_1 (and included in its bit-mask $b(n_1)$). The next ADV interval for this node is inferred from the end of the previous DATA transmission period and its ADV priority is set to *high*. Hence, it transmits a new ADV for TB i picking one of the *high* priority slots $1, 2, \ldots, \ell_{ADV}/2$ of this ADV interval, where this selection is made using the optimal access policy of [29] (which presents optimal slot selection probabilities, so as to minimize the probability of collision in the presence of multiple sending nodes). Within the same ADV interval, the *low* priority node n_o can transmit a further ADV exploiting slots $\ell_{ADV}/2 + 1, \ldots, \ell_{ADV}$, i.e., in the second half of the interval (using the same slot selection strategy). For our example, we assume that n_o sends a *low* priority ADV. At this point, two events can occur depending on the reception status of n_4:

- **C**ase 1: n_4 correctly receives the advertisement. This node will then transmit its REQ, asking for the last TB i and using one of the *high* priority slots $1, 2, \ldots, \ell_{REQ}/2$ of the next REQ interval (REQ priority always equals the priority of the corresponding ADV).
- **C**ase 2: n_4 does not receive the advertisement. In this case n_1 will not receive a response to its *high* priority ADV. Thus, it can still use one of the *low* priority slots $\ell_{REQ}/2 + 1, \ldots, \ell_{REQ}$ of the current REQ interval to respond to n_o's *low* priority ADV. Thanks to this fallback procedure, the current transmission *frame* will not go unused when there are no nodes providing advancement (referring to our example, when n_4 does not exist or its REQ is lost due to channel impairments).

As above, the slot selection for REQs adopts Tay's method [29] for both priority cases. We shall observe that our soft-TDMA scheme completely avoid collisions only during the DATA phases. In fact, according to our design the first two intervals of the time frame are accessed using slotted CSMA. Hence, the control packets sent during these intervals may collide. DATA packets, instead, are collision free because only one transmitter can be selected per neighborhood thanks to SYNAPSE++s ADV, REQ handshaking mechanism. In the following, we illustrate a few last technicalities:

4.3.5 ADV/REQ Suppression

Consider a node n and let TB i be the highest TB correctly received by this node. Upon receiving an ADV (REQ) advertising (asking for) TB j, this node cancels its ADV (REQ) transmission if and only if $j \geq i$ ($j > i$) and its current priority equals that of the received ADV (REQ).

4.3.6 Failure Management

Nodes which have not successfully decoded a given TB will ask for the same TB in the next transmission *frame*. However, they will always respond (REQ) to incoming ADVs using *low* priority slots and including the residual rank (number of additional linearly independent packets needed for $\mathbf{G}$ to have full rank) of their decoding matrix for this TB. The sender will then generate and transmit this number of additional packets. This is done to promote data correction through pipelining and thus limit the local retransmissions. Besides, this quickly pushes the data toward unexplored portions of the network.

4.3.7 Frame Synchronization

Upon the reception of the first valid ADV, a given node knows its own hop count as well as an estimate of the *frame* structure boundaries (i.e., it is synchronized at the *frame* level with its neighbors). We refer to this *frame* synchronization as "loose" as only those nodes within the same transmission range must be synchronized at the *frame* level for correct reception; in this case a precision of a few milliseconds suffices (time skews are accounted for adding guard intervals between ADV, REQ and DATA periods). However, nodes that are more than two hops apart can tolerate larger synchronization errors. In addition, each node adjusts its *frame* level boundaries upon the reception of an ADV either from a node closer to the originator or from a node having a larger hop count and advertising new TBs.

4.3.8 Back-Off Policy

A node that receives no response to its ADV defers the transmission of its next ADV according to a random timer whose maximum value is doubled at each transmission attempt (up to a maximum back-off time) and reset upon the reception of an REQ.

4.3.9 Energy Conservation

Due to the imposed *frame* structure, each node has a good knowledge of when data packets will be sent. This allows for the following optimizations: 1) a node can turn off its radio, thus conserving energy, whenever it detects the transmission of packets belonging to a TB that is not of interest for such node (this can be inferred from ADV and REQ periods as well as from the first packet of the DATA phase), 2) when a node does not detect any ADV or REQ messages it can additionally listen to the initial portion of the DATA period. Hence, it can still accept ongoing transmissions, in case these

are for missing TBs, or go to sleep otherwise. The optimization of these sleeping modes is left for future work.

5 Joint Pipelining and Fountain Codes Optimization

In this section, we further optimize the FC used in SYNAPSE++ in order to enhance the performance of its pipelining scheme. As outlined in Section 3.3, practical implementations of FC encoders/decoders require random number generators (RNG). The choice of the initialization seeds for these RNGs is particularly important. In fact, the correlation among the encoding vectors of different packets depends on how these seeds are picked. Specifically, a wrong selection of the seed leads to the transmission of linearly dependent encoded packets, and this impacts the performance in terms of decoding overhead. We remark that an attractive property of fountain codes lies in the fact that the coding process can be carried out without having any a priori information about the channel error probability. However, while this is true when packets are obtained through an ideal random number generator, pseudorandom number generators, due to the correlation that they inherently introduce in the packet stream, cause the performance of the decoding process to depend on the error rate. Nevertheless, the desirable features of FCs should not be altered by implementation details and the chosen seeds must lead to low decoding overhead regardless of the specific channel conditions.

In the following, we first describe our optimization of the seeds for a nonzero packet error probability and then we discuss a further optimization step to use FC in conjunction with pipelining. Carrying out this study on real sensor nodes would have been time prohibitive. Thus, in order to perform a significant number of experiments and characterize the performance of the seeds with high accuracy, we exactly reproduced the selected LFSR random generator in a simulator.

5.1 Seed Optimization against Channel Errors

For a given block length K, given the set of all possible seeds, R, and in the absence of channel errors ($p = 0$), we first identified the set R_0 that includes all seeds leading to a zero decoding overhead.[4] We then restricted our investigation to these seeds and characterized their performance over noisy channels estimating the decoding overhead for each of them. We considered independent packet losses over an erasure channel with packet error probability p.

Fig. 8 shows results for $\mathbb{E}[N]$, where N is the number of packets needed to successfully decode a given TB at the receiver side, as defined in Section 3.1. The solid line denotes the performance of the distribution selected in Section 3.2 (see Fig. 2) considering $p = 0$ and averaging over all seeds in R. In this case, $\mathbb{E}[N]$ is $\overline{N}(R) = 34.92$, which is slightly higher than that obtained for the same distribution in Section 3.2 ($\mathbb{E}[N] = 34.26$), due to the nonideality of the implemented pseudorandom generator. The dotted line shows the average performance for the seeds in R_0, referred to as $\overline{N}(R_0)$. The

4. For these experiments, we assumed $K = 32$ as this value fits well our memory requirements.

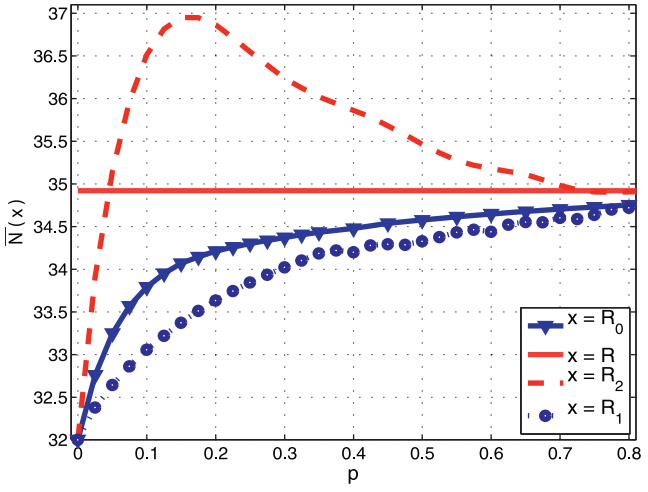

Fig. 8. $\mathbb{E}[N]$ versus packet error probability for different seed sets.

decoding overhead for these seeds increases for increasing p, approaching $\overline{N}(R)$ when $p \approx 0.8$, and this means that the choice of the seed is of little importance when p is very large. This behavior is expected as a large p will lead to many lost packets thus reducing the correlation among the few packets that are successfully delivered.

Since different seeds in R_0 do behave differently, we performed an exhaustive search over R_0. We identified a further set $R_1 \subset R_0$, comprising all seeds for which $\mathbb{E}[N] \leq \overline{N}(R_0)$. In principle, a different set R_1 should be obtained for each value of p, as the same seed might perform differently for different error rates. However, we noticed that, besides a small number of exceptions, seed behaviors were quite uniform, i.e., seeds that perform well when p is low usually perform well also at higher p. In other words, the ranking among seeds in terms of decoding performance is preserved as p varies.

Therefore, we empirically obtained R_1 for $p = 0.4$ as representative of all possible sets. After this, we verified that every seed in R_1 has in fact very low decoding overhead for a wide range of channel error probabilities. In Fig. 8, we show $E[N]$ for the seeds in R_1 and denote their average performance by $\overline{N}(R_1)$. In the same figure, we also show the performance ($\overline{N}(R_2)$) obtained from a third set R_2, including all seeds for which $\mathbb{E}[N] > \overline{N}(R_0)$.

5.2 Seed Selection for Pipelining

As a last optimization step, starting from set R_1, we selected a further subset $R^\star$ whose seeds also work properly for pipelining. Specifically, with reference to the scenario depicted in Fig. 5, hop-by-hop data dissemination techniques start transmitting to nodes in hop $k+1$ only when all nodes in hop k have completely received all data blocks. Instead, schemes using pipelining initiate the dissemination toward hop $k+1$ as soon as the first node in hop k decodes a valid data block (communication and synchronization issues were already addressed in Section 4.3).

A further advantage of pipelining is the possibility of correcting transmission errors (nodes within the current hop) while forwarding transport blocks toward the next hop. The key point of our seed design is that the seeds used at

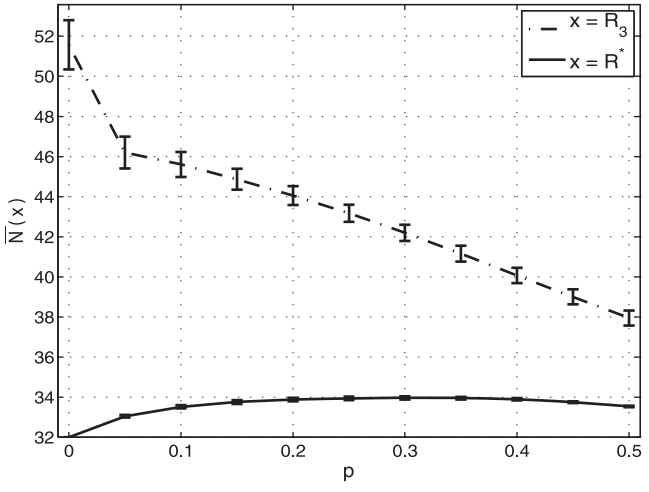

Fig. 9. $\mathbb{E}[N]$ versus packet error probability using seeds belonging to $R^\star$ and R_3 for $K' = 36$.

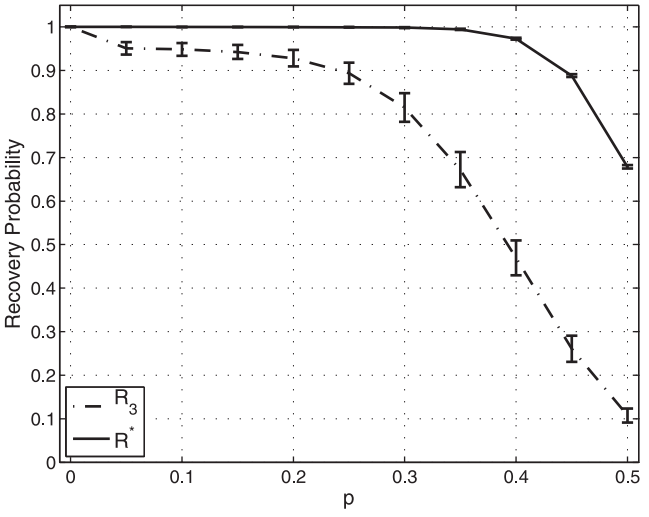

Fig. 10. Recovery probability versus p using seeds belonging to $R^\star$ and R_3 for $K' = 36$.

hop k and $k+1$ must be matched for this purpose. In detail, the transmission at hop k will be done using seed s_k. Referring to Fig. 5, imagine that nodes n_2 and n_3 are not able to decode the block while n_1 can correctly decode it. Now, according to the pipelining philosophy, n_1 will immediately transmit the block toward the next hop $k+1$, using seed s_{k+1} (instead of explicitly retransmitting data for the failed nodes n_2 and n_3). If s_{k+1} is carefully selected, unsuccessful nodes at hop k could correct their losses thanks to the data being forwarded toward hop $k+1$, i.e., in our example, n_2 and n_3 would simply append the new received packets (encoded using s_{k+1}) to their decoding matrix until it is invertible.

Following the above rationale, set R_1 has been extensively tested by quantifying the affinity of seed pairs (s_k, s_{k+1}) used over hops k and $k+1$. In particular, $\forall\ s_k, s_{k+1} \in R_1$, with $s_k \neq s_{k+1}$, we determined the *recovery probability* for a given node within hop k as follows:

- The transmission process starts at hop k sending a transport block of K' packets using seed s_k.
- We considered only those cases where at least one node within hop $k+1$ successfully decodes the TB and forward it using seed s_{k+1} (K' packets encoded using s_{k+1}).
- Conditioned on this, we thus computed the probability that any other node within hop $k+1$ can successfully decode the TB only using the two transmissions above ($2K'$ packets). This includes the nodes that were successful during the first transmission from hop k as well as those that recover the block by overhearing the second transmission.

We experimentally noticed that the behavior of seed pairs in terms of the above recovery probability only marginally depends on p. Also, only a few seeds in R_1 showed a significant performance loss when coupled with other seeds, while the large majority of them led to good multihop performance. Thus, a fourth seed set $R^\star \subset R_1$ was obtained by eliminating those few bad seeds from R_1. Set $R^\star$ is the final seed set that we use at every node to decode TBs. Note that all seeds in $R^\star$ lead to good recovery performance when used in combination over subsequent hops.

Examples of the gains achievable introducing this further optimization step are shown in Figs. 9 and 10, where we show $\mathbb{E}[N]$ (vertical bars indicate 95 percent confidence intervals), i.e., the average number of packets after which we can recover a TB, and the recovery probability for $R^\star$ and another set R_3 having the same cardinality and containing seeds randomly picked in R. For these graphs, we considered $K' = K + \delta = 36$. It should be observed that the optimizations carried out in this section have led to large improvements for both performance measures. For the seed selection policy, we opted for a very simple but effective approach. For any TB, all potential forwarders keep track of the seeds currently used within their neighborhood, storing them in the new set $R^\star_{\text{used}}$. Whenever a TB has to be sent, the sender randomly picks a seed in $R^\star \setminus R^\star_{\text{used}}$. $R^\star_{\text{used}}$ is emptied when the node operates on a new TB or $R^\star = R^\star_{\text{used}}$.

We conclude this section with some considerations on the cardinality of the different sets that were identified. Working with words of 16 bits means that R contains $2^{16} - 1 = 65{,}535$ seeds. For computational reasons, we restricted our study to 5,000 seeds, randomly picked in this set. About 10 percent of them, i.e., slightly more than 500 seeds, belong to R_0. The cardinality of R_1 is $|R_1| = 50$ and $R^\star$ has 35 elements meaning that, on average and for the considered RNG, only 1 every 150 seeds possesses all the required properties.

6 Experimental Results

In this section, we present the results of the experimental comparison between SYNAPSE++ and Deluge in a real multihop deployment. Aiming at a fair comparison between the two schemes we proceeded as follows:

- at the beginning of the experiments we programmed each sensor node with an application. For different experiments the only characteristic that we change in this initial application is the MAC protocol used. We tested the behavior of the dissemination applications with a low power listening (LPL) MAC [30] for three different configurations: 1) LPL disabled (i.e.,

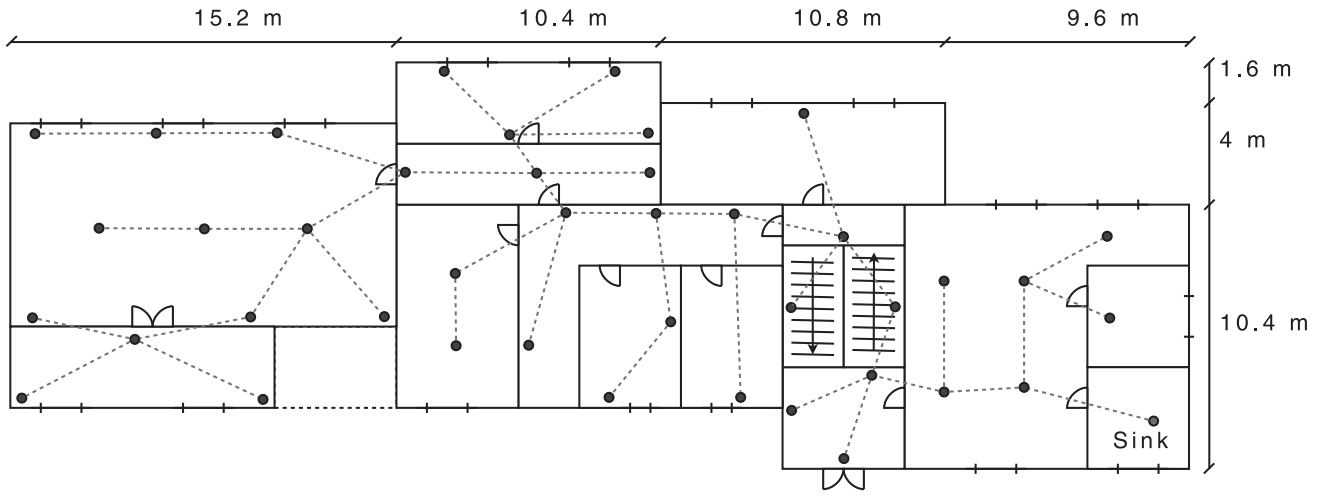

Fig. 11. Map of the WSN deployment.

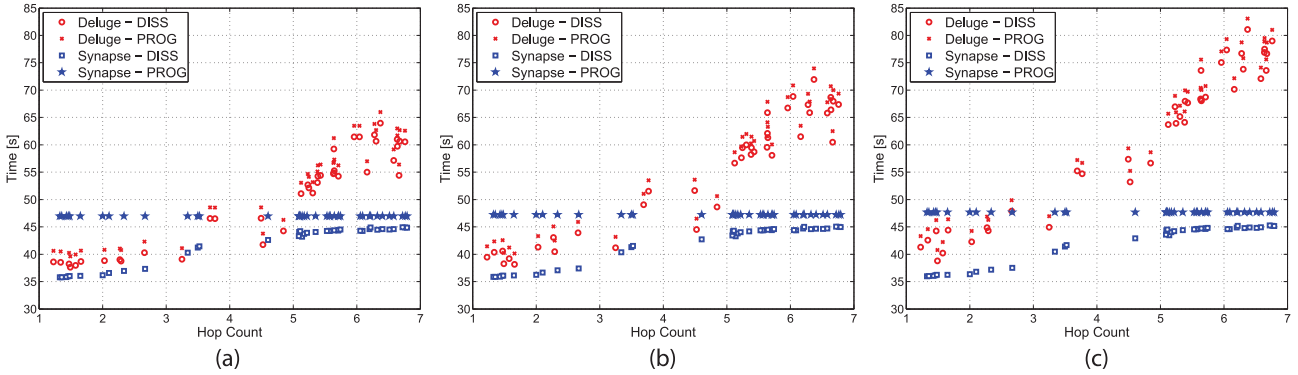

Fig. 12. Dissemination (DISS) and programming time (PROG) as a function of the hop count distance. (a) Duty cycle 100 percent. (b) Duty cycle 7 percent. (c) Duty cycle 2.5 percent.

radio always on), 2) LPL enabled with duty cycles of 7 percent, and 3) of 2.5 percent,

- during each experiment we disseminate the same application that was originally loaded on the motes, with the same MAC policy. We repeated this dissemination procedure several times,
- we measured the following performance metrics: 1) *dissemination time*, time elapsed between the instant when a new dissemination command is issued by the sink node to the instant when the program image is correctly received by all nodes, and 2) *programming time*, time elapsed between the instant when a new dissemination command is issued at the sink node to the instant when the program image is correctly running at all nodes.

Exactly the same amount of data (i.e., a program image of 11.5 kB) was transmitted by the two dissemination protocols in all experiments.

6.1 WSN Deployment

We deployed 42 sensor nodes on the first floor of a three stories industrial building, including offices and commercial spaces. Our sensors spanned an area of about 15×46 square meters. We placed a sensor in each office, up to nine within larger commercial spaces, and we positioned a few further sensors on stairs and hallways to keep the structure connected. Overall, this led to a network topology with maximum path lengths of 14 hops. Fig. 11 shows the layout of our deployment: the eastmost node has been chosen as the dissemination initiator (referred to as *sink*).

The hardware platform consists of TmoteSky wireless nodes (same hardware as Crossbow TelosB [31] sensor nodes). These sensors are equipped with an IEEE 802.15.4 compliant radio transceiver (a ChipCon CC2420 radio chip working in the 2.4-GHz frequency band), an MSP430 16 bit microcontroller with 10 kB on chip RAM, 48 kB FLASH/ROM and 1,024 kB external FLASH memory. The radio chip has a nominal data rate of 250 kB/s. For SYNAPSE++, we split the application into blocks of $K = 32$ packets, having data and header fields of 25 and 15 bytes, respectively, and considered transport blocks of $K' = K + \delta = 36$ packets, where $\delta = 4$ is the redundancy per block that we added to cope with channel errors. ADV and REQ messages take 25 and 14 bytes, respectively. For the packet structure, we opted for the standard `AM message` of TinyOS, adding 4 bytes for block id and random seed to the usual 11 bytes of header. We remark that both Deluge and SYNAPSE++ were designed to be 100 percent reliable so that all nodes in the network correctly receive every byte of the transmitted image, and this has been confirmed by all of our experiments. Thus, in this section, we only look at the dissemination time and number of packets sent.

6.2 Experimental Results

Fig. 12 shows the dissemination (DISS) and programming time (PROG) for SYNAPSE++ and Deluge for the described

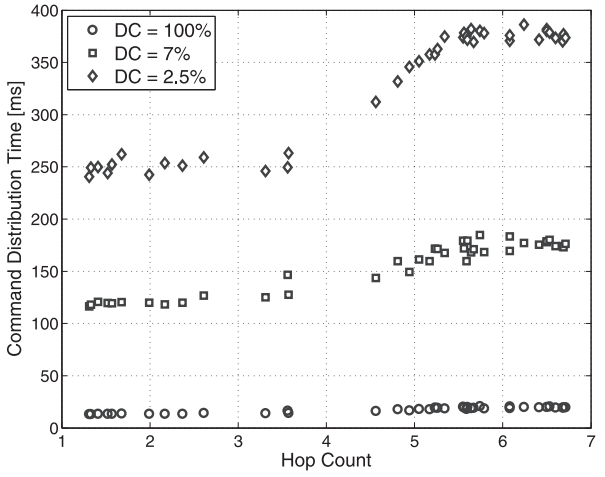

Fig. 13. Command distribution time: time taken to distribute the `reload` SYNAPSE++ command to all nodes for different values of the duty cycle (DC) at the MAC.

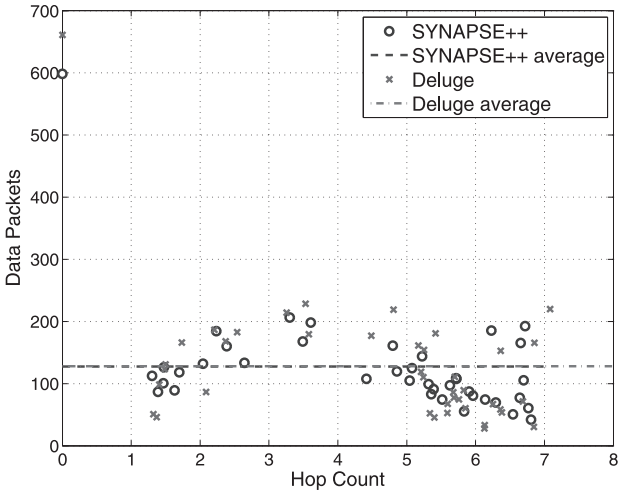

Fig. 14. Data packets transmitted as a function of the hop count distance.

WSN deployment. Each point in the graphs represents the performance of a single node; its abscissa is the maximum hop count distance from the sink during a run of the dissemination experiment, averaged over 100 runs (the same number of experiments was considered for the remaining results in this section). The dissemination time of SYNAPSE++ is the sum of two contributions: 1) `reload` SYNAPSE++, the time required to distribute a reboot command to all nodes and subsequently load SYNAPSE++; 2) `disseminate` application, the time necessary to disseminate the new program image to all nodes. Note that the initial command distribution protocol must be integrated in the original application running on nodes and depends on the specific implementation of it. For our experiments, we used a simple flooding: even though more complicated and reliable approaches are possible, this technique always reached all nodes.

In Fig. 13, we show the time required to distribute the `reload` command to all sensors. The distribution of any other command when SYNAPSE++ is running takes the same amount of time as that in Fig. 13 for a duty cycle of 100 percent. Upon the completion of the command distribution phase the nodes reboot with SYNAPSE++, and this takes a constant time, which is the same for SYNAPSE++ and Deluge and is approximatively equal for all nodes. The second contribution to the dissemination time is the time taken for the actual dissemination of the code image, which is achieved using the algorithm in Section 4.3.

From Figs. 12a, 12b and 12c we observe that the dissemination time of Deluge is dramatically influenced by the MAC protocol implemented in the application originally loaded on the motes. This is because Deluge runs in parallel with the application and exploits the same LPL MAC to disseminate the program image. The LPL technique works fine with low-rate traffic shape, which might be the case for the regular behavior of the application but not for the intensive transmission activity required by the dissemination phase. Specifically, with short duty cycles longer preambles must be used by the LPL MAC wakeup procedure upon the transmission of control and data packets, which become very frequent during the dissemination operation. The programming time of Deluge is given by the dissemination time plus a constant additional delay due to the reboot of the nodes.[5] We note that the dissemination time of SYNAPSE++ scales better as a function of the hop count distance and is not impacted by the duty cycle of the preloaded LPL MAC. In fact, according to SYNAPSE++'s design (see Section 4.1) the preloaded application and its MAC are replaced with SYNAPSE++ before initiating the dissemination procedure. Hence, the actual reprogramming phase is performed using SYNAPSE++'s soft-TDMA (see Section 4.3).

The programming time of SYNAPSE++ is obtained through a back-propagation scheme where nodes recursively inform their parents about their programming status. In detail, the reprogramming procedure ends at a given node whenever this node as well as all its neighbors with higher hop count have received the application. In this case, this node will send an ADV with a completion bit set and, to speed up the notification process, these final ADVs are sent using a simple CSMA. Note that soft-TDMA and the related slot structure is not applied here as in this last phase no further data are transmitted.

Points in Figs. 14 and 15 represent the average performance of specific nodes over all experiments, whereas lines correspond to the average performance over all nodes and experiments. In Fig. 14, we show the number of data packets sent during a reprogramming operation. For this metric, SYNAPSE++ attains similar performance with respect to Deluge. The points having hop count zero represent the data packets transmitted by the sink. These values are considerably higher than the remaining ones as the sink must transmit all transport blocks, whereas other sensor nodes take turns in transmitting, thus effectively sharing the traffic load.

Fig. 15 refers to the number of control packets sent during a dissemination. SYNAPSE++ performs slightly better than Deluge in terms of control overhead. Again,

5. The results for Deluge have been obtained using the disseminate and reboot `-dr` command.

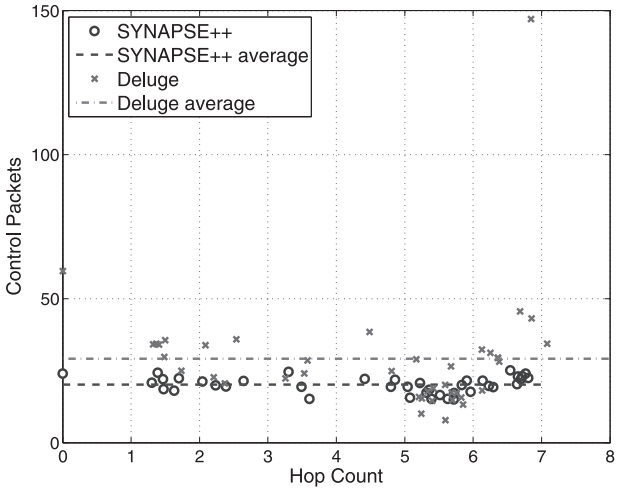

Fig. 15. Control packets transmitted as a function of the hop count distance.

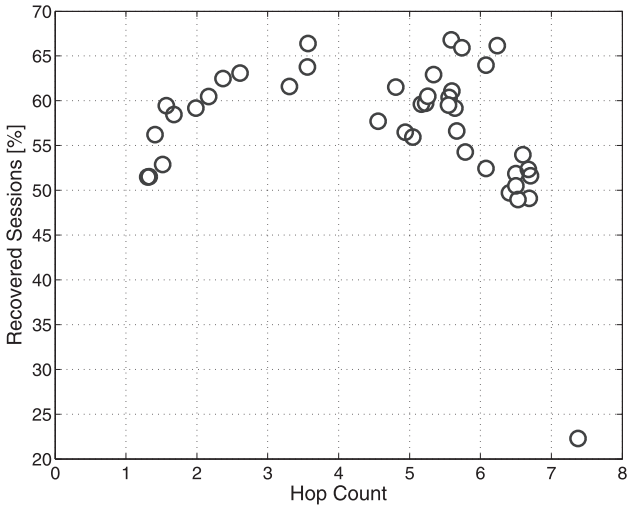

Fig. 16. Effectiveness of overhearing.

the rightmost node in the figure has the largest performance gap; this is due to its poor link conditions and to the fact that SYNAPSE++ is more efficient in this case. Also, for SYNAPSE++ this metric has a smaller dispersion around its mean. The results in Figs. 14 and 15 were obtained for a duty cycle of 100 percent. We however obtained very similar results for other duty cycles. In fact, SYNAPSE++ is not impacted by the application's MAC, whereas for Deluge the LPL MAC only leads to longer transmission phases, without changing the protocol dynamics.

Finally, Fig. 16 shows the recovery probability during pipelining, calculated as the fraction of times a node did not successfully receive the first transmission of a transport block and could successfully recover it in the next dissemination phase overhearing some other node's transmission.[6] If this recovery does not succeed, the node in question will explicitly ask for the retransmission of the missing blocks. Our pipelining was however designed in order to take the maximum advantage of overheard transmissions, so as to avoid further explicit retransmission phases. From the experimental results of Fig. 16, we see that overhearing is effective for 50 percent to 70 percent of the dissemination sessions, while only the node at the last hop (the rightmost point in the plot) has poor performance. This is due to the fact that this node could not overhear forwarded packets as it did not have neighbors at the same hop level. These results confirm the validity of SYNAPSE++'s pipelining scheme.

In conclusion, we observe that SYNAPSE++ effectively exploits pipelining in multihop environments, where the soft-TDMA strategy together with FCs give some advantages in terms of dissemination time and control overhead. Also, SYNAPSE++'s design allows us to fully decouple the reprogramming application from the existing software on the nodes. This, as shown by our results for the low-power listening MAC, prevents the performance of SYNAPSE++ from getting impacted by a suboptimal or faulty design of application components.

6. Note that the *recovery probability* as defined in Section 5 is higher than that of Fig. 16 as it also includes those nodes that received the block correctly during its first transmission.

7 User Experience and Conclusions

We now conclude the paper by discussing our experience at the IBM T.J. Watson research facility in New York, where we used SYNAPSE++ as a support tool for the SEAIT project [32]. SEAIT is a WSN system designed specifically to address the requirements of visibility and monitoring of the railroad system. A testbed for SEAIT was set up by deploying a number of sensor boards on the rooftop of the IBM facility, with distances among sensors comparable to those at which nodes will be likely to work in the actual deployment on trains. This testbed consists of a network covering a quarter of a mile with a maximum path length of seven hops. As expected, a number of problems that did not arise during the indoor experiments arose in the outdoor deployment and required several code updates. SYNAPSE++ greatly simplified the reprogramming procedure, reducing the time from the 2 hours required to manually reprogram the nodes to the few minutes required to distribute the application wirelessly. The SEAIT prototype was implemented in TinyOS 1. Although SYNAPSE++ is written in TinyOS 2, the two applications could coexist without any problems. As per our discussion in Section 4.2, SEAIT did not require any change but the implementation of a *reboot* function.

The experience at IBM research allowed us to tune SYNAPSE++ so that the application is now robust to several hardware problems. For example, CRC fields were added to each transport block to identify erroneous data due to faulty portions of the FLASH memory. The feedback that we obtained from this first use of SYNAPSE++ in support to another research team is extremely positive. The system addresses their requirements, showing good stability and usability, and physical access to the testbed is now only needed to turn off the nodes or to change their batteries.

To conclude, decoupling the reprogramming software from the application has been shown to be beneficial as it shields the reprogramming procedure from side effects due to application components, such as aggressive sleeping behavior or faulty MAC design. Also, this design permits the dissemination of applications even when they cannot run concurrently with the reprogramming software due to,

e.g., copyright, code or memory issues. Overall, from our performance evaluation as well as from the above experience we can conclude that SYNAPSE++ is effective in providing fast and reliable dissemination of code images in multihop WSNs.

ACKNOWLEDGMENTS

This material is based upon work partially supported by the European Commission under contract number INFSO-ICT-215923 (SENSEI), by the Italian Foundation Cassa di Risparmio di Padova e Rovigo (CARIPARO), by the Italian Ministry of University and Research under the International FIRB program, grant no. RBIN047MH9, and by the Swedish Agency for Innovation Systems (VINNOVA). A previous version of this paper was presented at IEEE SECON 2008.

REFERENCES

[1] J.W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proc. ACM SenSys,* Nov. 2004.
[2] S.S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS),* June 2005.
[3] 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.
[4] R.K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *Proc. IEEE INFOCOM,* May 2007.
[5] J. Nonnenmacher, E.W. Biersack, and D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission," *IEEE/ACM Trans. Networking,* vol. 6, no. 4, pp. 349-361, Aug. 1998.
[6] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A.F. Harris III, and M. Zorzi, "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes," *Proc. IEEE Comm. Soc. Conf. Sensor, Mesh and Ad Hoc Comm. and Networks (SECON),* June 2008.
[7] 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.
[8] J. Jeong, S. Kim, and A. Broad, "Network Reprogramming," http://www.tinyos.net/tinyos-1.x/doc, Aug. 2003.
[9] T. Stathopoulos, J. Heidemann, and D. Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," CENS Technical Report 30, 2003.
[10] V. Naik, A. Arora, P. Sinha, and H. Zhang, "Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices," *Proc. IEEE Real-Time Systems Symp. (RTSS),* Dec. 2005.
[11] S.-J. Park, R. Sivakumar, I. Akyildiz, and R. Vedantham, "GARUDA: Achieving Effective Reliability for Downstream Communication in Wireless Sensor Networks," *IEEE Trans. Mobile Computing,* vol. 7, no. 2, pp. 214-230, Feb. 2008.
[12] L. Huang and S. Setia, "CORD: Energy-Efficient Reliable Bulk Data Dissemination in Sensor Networks," *Proc. IEEE INFOCOM,* Apr. 2008.
[13] C.-J.M. Liang, R. Musăloiu-Elefteri, and A. Terzis, "Typhoon: A Reliable Data Dissemination Protocol for Wireless Sensor Networks," *Proc. European Conf. Wireless Sensor Networks (EWSN),* Jan. 2008.
[14] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes," *Proc. IEEE Int'l Conf. Information Processing in Sensor Networks (IPSN),* Apr. 2008.
[15] 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,* Apr. 2008.
[16] D.J.C. MacKay, "Fountain Codes," *IEE Proc. Comm,* vol. 152, no. 6, pp. 1062-1068, Dec. 2005.
[17] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks," *Proc. ACM SenSys,* Nov. 2006.
[18] "Contiki: The Operating System for Embedded Smart Objects—The Internet of Things," http://www.sics.se/contiki, Mar. 2009.
[19] P.J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks," *Proc. European Conf. Wireless Sensor Networks (EWSN),* Feb. 2006.
[20] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient Sensor Network Reprogramming through Compression of Executable Modules," *Proc. IEEE Comm. Soc. Conf. Sensor, Mesh and Ad Hoc Comm. and Networks (SECON),* June 2008.
[21] M. Luby, "LT Codes," *Proc. 43rd Ann. IEEE Symp. Foundations of Computer Science,* Nov. 2002.
[22] D.J.C. MacKay, *Information Theory, Inference, and Learning Algorithms.* Cambridge Univ. Press, 2003.
[23] M. Rossi, M. Zorzi, and F.H. Fitzek, "Link Layer Algorithms for Efficient Multicast Service Provisioning in 3G Cellular Systems," *Proc. IEEE Global Comm. Conf. (Globecom),* Nov. 2004.
[24] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical Loss-Resilient Codes," *Proc. 29th Ann. ACM Symp. Theory of Computing,* May 1997.
[25] E. Hyytiä, T. Tirronen, and J. Virtamo, "Optimizing the Degree Distribution of LT Codes with an Importance Sampling Approach," *Proc. Sixth Int'l Workshop Rare Event Simulation (RESIM '06),* Oct. 2006.
[26] D.E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms,* third ed. Addison-Wesley, 1997.
[27] "TinyOS: An Open Source OS for the Networked Sensor Regime," http://www.tinyos.net, Mar. 2009.
[28] L. Badia, N. Bui, M. Miozzo, M. Rossi, and M. Zorzi, "On the Exploitation of User Aggregation Strategies in Heterogeneous Wireless Networks," *Proc. IEEE Int'l Workshop Computer-Aided Modeling, Analysis and Design of Comm. Links and Networks (CAMAD),* June 2006.
[29] Y.C. Tay, K. Jamieson, and H. Balakrishnan, "Collision-Minimizing CSMA and Its Applications to Wireless Sensor Networks," *IEEE J. Selected Areas in Comm.,* vol. 22, no. 6, pp. 1048-1057, Aug. 2004.
[30] J. Polastre, J.L. Hill, and D.E. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks," *Proc. ACM SenSys,* Nov. 2004.
[31] "Crossbow," http://www.xbow.com, Mar. 2009.
[32] J.M. Reason and R. Crepaldi, "Ambient Intelligence for Freight Railroads," *IBM J. Research and Development,* vol. 53, no. 3, May/June 2009.

Michele Rossi received the laurea degree (with honors) in electrical engineering and the PhD degree in information engineering from the University of Ferrara in 2000 and 2004, respectively. From March 2000 to October 2005, he has been a research fellow in the Department of Engineering, University of Ferrara. During 2003, he was on leave at the Center for Wireless Communications (CWC) at the University of California San Diego (UCSD), where he performed research on wireless sensor networks. In November 2005, he joined the Department of Information Engineering, University of Padova, Italy, where he is currently an assistant professor. He is currently part of the EU-funded SENSEI project and of the WISE-WAI project, both on wireless sensor networks. His research interests are centered around the dissemination of data in distributed ad hoc and wireless sensor networks, including integrated MAC/routing schemes, data dissemination via network coding, the application of compressive sensing techniques for the reconstruction of signals in wireless sensor networks and cooperative routing policies for ad hoc networks. He is a member of the IEEE.

Nicola Bui received the laurea degree in information engineering in 2003 and the specialistic laurea degree in telecommunication engineering in 2005, both from the University of Ferrara. He is currently a general manager at Patavina Technologies, a spin-off of the University of Padova, operating in the ICT field. He has been a research fellow with Consorzio Ferrara Ricerche (CFR), Italy, and with the Department of Information Engineering (DEI), University of Padova, for four years. During this period, he has been involved in three European funded projects: Ambient Networks on heterogeneous networks and e-SENSE and SENSEI on wireless sensor networks. His main research interests include the design, simulation, and experimentation of protocols and applications for wireless sensor networks and embedded systems.

Giovanni Zanca received the laurea degree in telecommunication engineering from the University of Padova in 2007. He is currently with the Department of Information Engineering, University of Padova, Italy. His main research interests include the design and the performance evaluation of localization systems for ad hoc networks and the implementation of algorithms on wireless sensor networks.

Luca Stabellini received the BS degree in electrical engineering and the MS degree in electrical and telecommunication engineering (both with honors) from the University of Ferrara, Italy, in 2004 and 2006, respectively. From March to September 2006, he completed a six-month research internship at Thales Research and Technology, Palaiseau, France. Since October 2006, he has been with the Radio Communication Systems Department at the Royal Institute of Technology (KTH), Stockholm, Sweden, working toward the PhD degree. His current research interests include the design of interference mitigation and interference avoidance techniques for wireless sensor networks with special emphasis on energy-efficient and simple dynamic spectrum access schemes.

Riccardo Crepaldi received the MS (laurea) degree in telecommunications engineering from the University of Padova, Italy, in 2006. He is currently working toward the PhD degree in the Computer Science Department, University of Illinois at Urbana-Champaign, under the supervision of Professor Robin Kravets. After his graduation, he was a research scientist in the Signet Group, University of Padova, until 2007. He was engaged in research on wireless sensor networks and designed and deployed a WSN testbed and developed management tools for it. He also worked on the design and performance analysis of routing and localization algorithms for WSNs. The scope of his research involves systems design and management and service discovery for wireless sensor networks and delay-tolerant networks. He is a student member of the IEEE.

Michele Zorzi received the laurea degree and the PhD degree in electrical engineering from the University of Padova, Italy, in 1990 and 1994, respectively. During the academic year 1992-1993, he was on leave from the University of California, San Diego (UCSD), attending graduate courses and doing research on multiple access in mobile radio networks. In 1993, he joined the faculty of the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy. After spending three years with the Center for Wireless Communications at UCSD, in 1998, he joined the School of Engineering, University of Ferrara, Italy, and in 2003, he joined the Department of Information Engineering at the University of Padova, Italy, where he is currently a professor. His current research interests include performance evaluation in mobile communications systems, random access in mobile radio networks, ad hoc and sensor networks, energy-constrained communication protocols, and cognitive radio and networks. He was the editor-in-chief of the *IEEE Wireless Communications Magazine* from 2003 to 2005, is currently the editor-in-chief of the *IEEE Transactions on Communications*, serves on the steering committee of the *IEEE Transactions on Mobile Computing*, and serves on the editorial boards of the *Wiley Journal of Wireless Communications and Mobile Computing* and the *ACM/URSI/Kluwer Journal of Wireless Networks*. He was also a guest editor for special issues of the *IEEE Personal Communications Magazine* (energy management in personal communications systems) and the *IEEE Journal on Selected Areas in Communications* (multimedia network radios). He is a fellow of the IEEE.