SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks using Fountain Codes

Michele Rossi[†], **Giovanni Zanca**[†], **Luca Stabellini**[⋆],
Riccardo Crepaldi[‡], **Albert F. Harris III**[‡] and **Michele Zorzi**[†]

[†]Dept. of Information Engineering, University of Padova, 35131 Padova, Italy.
Email: {rossi, zancagio, zorzi}@dei.unipd.it
[‡]Dept. of Computer Science, University of Illinois, Urbana-Champaign, 61801 Illinois, USA.
Email: {rcrepal2, aharris}@uiuc.edu
[⋆]Wireless@KTH, Royal Institute of Technology, Electrum 418, SE-164 40 Kista, Sweden.
Email: luca.stabellini@radio.kth.se

Abstract*—Wireless reprogramming is a key functionality in Wireless Sensor Networks (WSNs). In fact, the requirements for the network may change in time, or new parameters might have to be loaded to change the behavior of a given protocol. In large scale WSNs it makes economical as well as practical sense to upload the code with the needed functionalities without human intervention, i.e., by means of efficient over the air reprogramming. This poses several challenges as wireless links are affected by errors, data dissemination has to be 100% reliable, and data transmission and recovery schemes are often called to work with a large number of receivers. State of the art protocols, such as Deluge, implement error recovery through the adaptation of standard Automatic Repeat reQuest (ARQ) techniques. These, however, do not scale well in the presence of channel errors and multiple receivers. In this paper, we present an original reprogramming system for WSNs called SYNAPSE, which we designed to improve the efficiency of the error recovery phase. SYNAPSE features a hybrid ARQ (HARQ) solution were data are encoded prior to transmission and incremental redundancy is used to recover from losses, thus considerably reducing the transmission overhead. For the coding, digital Fountain Codes were selected as they are rateless and allow for lightweight implementations. In this paper, we design special Fountain Codes and use them at the heart of SYNAPSE to provide high performance while meeting the requirements of WSNs. Moreover, we present our implementation of SYNAPSE for the Tmote Sky sensor platform and show experimental results, where we compare the performance of SYNAPSE with that of state of the art protocols.

I. Introduction

Many applications currently exploit wireless sensor networks (WSNs) for long term data gathering, ranging from environmental sensing, manufacturing plant control, etc, and many more are under development. We note that the requirements for the network (which translate into functionalities to support) may change in time. Also, the WSN itself might be moved to a different place thus requiring reconfiguration. Finally, we might want to reconfigure on the fly a given protocol through, e.g., the upload of new specifications for its rules and general behavior. These needs all call for energy efficient, scalable, topology independent and fast methods to wirelessly reprogram the WSN. In order to reach these goals, a protocol must meet several requirements which are peculiar to WSNs. First, it is crucial that the code delivery is 100% reliable and reaches all intended destination nodes. It shall be so regardless of channel errors, link variability and topology changes. Second, program sizes can be as large as 48 Kbytes, usually packets are 26 bytes long and sensor nodes have a limited amount of RAM (4 or 10 Kbytes, depending on the sensor hardware) and FLASH memory (usually 512 Kbytes). This means that the code cannot be entirely stored in RAM, i.e., we are dealing with large data transfers if compared with the actual memory capabilities of the sensors: as a result, dedicated dissemination/Automatic Repeat reQuest (ARQ) schemes are to be designed. Third, the WSN environment is inherently multi-hop, which implies that special protocols are needed to ensure reliable dissemination over multiple hops without requiring any a priori knowledge about the network topology. Fourth, WSNs are usually highly populated with wireless devices, thus if no proper countermeasures are taken, it is likely that many senders will transmit at the same time. This will translate into collisions, that result in an overall slow-down of the delivery process as well as decreased energy efficiency. Therefore, special algorithms are needed to properly handle the selection of senders and intelligent schemes for feedback suppression (e.g., ARQ NACKs) shall be implemented to reduce collisions [1].

A few practical algorithms have been designed to solve the above problems. The state-of-the-art is represented by the following four protocols: Deluge [2], MNP [3], Stream [4] and Freshet [5]. These schemes all transfer data in chunks (referred to as *pages*) in multi-hop WSNs. Some form of epidemic routing (all), intelligent sender election (MNP) as well as transmission/feedback suppression (all), pipelining (all) and aggressive sleeping behavior (MNP and Freshet) have been used. However, we observe that the transmission of *pages* and the subsequent error recovery is always obtained through the adaptation of standard ARQ techniques. That is, the erroneous reception of part(s) of the code is notified through some sort of status messages (basically NACKs, even though bit-masks can be used for improved efficiency see, e.g., [3]), which are sent to the sender upon the completion of their transmission.

In this paper we present SYNAPSE, an original protocol for reprogramming WSNs. While incorporating many of the above techniques, SYNAPSE *adopts an extremely efficient (and different) data transmission/recovery paradigm*. In fact, a Fountain Code [6] (FC), specifically designed to meet the needs of sensor

network reprogramming, is used at the heart of the data dissemination/recovery process. This code is designed to maintain a high efficiency, in terms of overhead, in the face of small packet sizes and typical program lengths. These codes were selected due to their desirable properties: FCs are rateless and have a low computational complexity, as encoding and decoding are performed efficiently through XOR operations. Our Fountain Code has been implemented on Tmote Sky nodes and shown to execute efficiently even with the limited available processing power. Our experiments show that we achieve reliable network programming with very low overhead compared to other current in-network reprogramming techniques [2]. It shall be observed that our present research work is complementary in nature to what previously done: while others mainly concentrated their study upon devising smart algorithms (i.e., modified epidemic schemes) for sender selection, sleeping modes etc., our focus is on extremely efficient solutions for the local delivery of the data (i.e., between the senders and their neighbors), as well as their proper integration with previous techniques.

The rest of the paper is organized as follows. Section II surveys related work. Section III describes the structure and the algorithms used in SYNAPSE. Section IV presents the design of the Fountain Code we adopt in our framework as well as its performance. Section V shows our experimental results, where the performance of SYNAPSE is compared with that of state-of-the-art algorithms. Finally, Section VI concludes the paper.

II. Related Work

XNP [7] is the first network reprogramming protocol for WSNs. It operates only over a single hop and does not support incremental updating of the program image. The Multihop Over the Air Protocol (MOAP) [8] extended the code delivery to multi-hop networks. It introduced some interesting features for the local recovery of data (NACKs, local broadcast, sliding window recovery), which are all used by the most recent protocols. MOAP disseminates data in a hop-by-hop fashion, i.e., a node has to receive the whole program before starting the dissemination over the next hop.

Next, we discuss the four protocols that define the state-of-the-art for wireless sensor network reprogramming: Deluge [2], MNP [3], Freshet [5] and Stream [4]. Deluge disseminates the code in multi-hop environments exploiting an epidemic routing algorithm, which uses a three-way handshake based on advertisement (ADV), request (REQ) and actual code (CODE) transfer. Note that ADVs and REQs have smaller sizes if compared to data packets; this reduces the transmission overhead when nodes contend for the channel. In Deluge, the code is subdivided into pages, which are disseminated using a NACK-based ARQ protocol. The code is transmitted, page by page, via broadcast and pipelining is implemented. Pipelining allows a node that correctly receives a page from a node within its previous hop to promptly start the dissemination of this page to the next hop. The randomization of the transmission of the advertisement within predetermined time windows as well as advertisement suppression are implemented to reduce the congestion in the propagation of the code through multiple hops. MNP [3] has many features in common with Deluge. In addition, it implements special algorithms to reduce the problems due to collisions and hidden terminals. This is achieved through a distributed priority assignment so that, in a neighborhood, there is at most one sender transmitting the program at any given time. The sender election is greedy and distributed, i.e., there is no need to know the topology in advance. In MNP the senders with a higher number of potential receivers are assigned higher priority and sleeping modes are also used to reduce the energy consumption; a sender can go to sleep when a neighbor with higher priority has data to send. Freshet [5] is based on Deluge and aggressively optimizes the energy consumption during reprogramming. In an initial phase, some meta-data (information) about the code 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 topology information, nodes estimate when the code will actually get to their vicinity and enter a sleeping period accordingly. Some other features, such as the dynamic adjustment of the frequency at which meta-data is transmitted, are implemented as well. Stream [4] builds on Deluge and optimizes *what* is actually sent over the channel. Common intuition would be to transfer only what actually needed, i.e., the *program image*. However, Deluge, MNP and Freshet all disseminate the image of the programming protocol together with that of the program to be transferred. This considerably inflates the amount of data to be disseminated (up to 20 folds for the transmission of a program image consisting of a single page [4]). Stream obviates this problem by pre-installing in each sensor node, before its actual deployment, the re-programming application. This is done through the segmentation of the FLASH into multiple partitions so that the re-programming 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 re-programming image together with the actual program image. In reference [4] this strategy is implemented on Deluge and is shown to provide substantial performance improvements.

SYNAPSE adopts many of the above techniques. It uses three way handshakes as per the ADV-REQ-CODE paradigm introduced above. It implements randomization when sending advertisements. It exploits broadcast transmissions for the code and NACKs to request missing data and it implements the method proposed in Stream [4]. On the other hand, it features new elements such as the extension of Deluge's FLASH memory partition management (see `PartitionManager` in Section III) as well as a novel hybrid ARQ error recovery mechanism.

III. A Data Dissemination System for Wireless Sensor Networks

In the following Section III-A we illustrate SYNAPSE's architecture. Section III-B introduces the `BootLoader`, which we realized to allow the management of the FLASH (formatting, partitioning, etc.) and load new programs. In Section III-C we present the Fountain Based dissemination protocol.

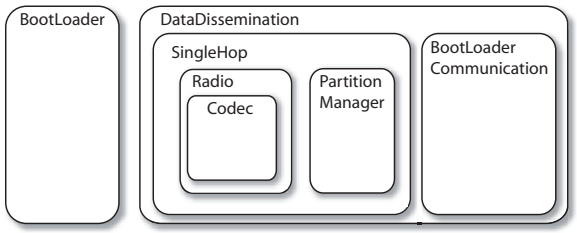

Fig. 1. SYNAPSE's component architecture in TinyOS-2.x.

A. General Architecture

In what follows, we describe the SYNAPSE data dissemination system by discussing its main functional blocks.

Structure of the software: the software was developed in a modular and portable way to facilitate its extension or the modification of any of its parts. The software architecture is shown in Fig. 1. There are two independent macro-blocks: `BootLoader` and `DataDissemination`, which cannot be executed concurrently as the micro-controller is single-task. The communication between `BootLoader` and `DataDissemination` is possible thanks to the *Information Memory*, a portion of the internal FLASH, which is preserved even after a device reset.

BootLoader: Our sensor nodes feature a TI MSP430 micro-controller with direct access to an internal FLASH of 48 Kbytes; additional storage is provided by an external FLASH of 1024 Kbytes. The `BootLoader` is loaded at boot time, and handles read and write operations between these memories. It can copy applications into the external FLASH and load them on demand. Due to its importance, we present the `BootLoader` in greater detail in Section III-B.

BootLoader Communication: this is a TinyOS module allowing the communication between TinyOS applications (in our case the `DataDissemination` module) and the `BootLoader`. It is implemented to hide hardware details. This module provides a *reboot* command which is used to reset the device. This command is executed when a new application is received and needs to be loaded in place of the current program.

DataDissemination: the `DataDissemination` module communicates directly with the `SingleHop` and the `BootLoaderCommunication` modules. This is the only module that has to be included in a TinyOS project to support in-network reprogramming. `DataDissemination` implements an epidemic routing algorithm as well as a three-way-handshake similar to the one in [2].

SingleHop: `DataDissemination` uses the `SingleHop` module to send/receive data to/from the devices in the current node's neighborhood. Hence, `SingleHop` manages local transmissions, whereas `DataDissemination` decides whether or not the current node should contribute itself to the data dissemination, by initiating a local dissemination procedure. When a node actively participates in the dissemination, the `SingleHop` module reads *transport blocks* of data from the external FLASH and sends them to the `Radio` module.

Radio: it is responsible for transmitting and receiving data packets. For improved efficiency, the `Radio` module implements a Hybrid ARQ (HARQ) strategy, where packets are encoded according to a digital fountain approach [6]. These issues are discussed in greater detail in Sections III-C and IV. The `Radio` module provides the `Codec` with the current set of original packets (the current *transport block*), which are used by the `Codec` to obtain encoded packets. These are then passed to the `Radio` module for their actual transmission. To summarize, the main functionalities of the `Radio` module are: 1) transmission and reception of encoded data packets, 2) implementation of a HARQ retransmission strategy, 3) control of the `Codec`, determining how many packets are to be encoded, when incremental redundancy has to be created, etc.

Codec: the codec implements the Fountain Code coding routines, which were specifically designed for sensor devices, see Section IV for further details.

Partition Manager: it is a TinyOS module we wrote to provide functionalities for reading, writing and creating memory partitions in the external FLASH. The FLASH is partitioned and used as an external disk through the definition of a partition table, which is stored at the beginning of the memory. This allows for a hardware independent approach which facilitates the porting to a new type of memory chip. More details are given in the next section.

B. Boot Loader for Wireless Network Reprogramming

The `BootLoader` can copy applications between internal and external FLASH memories and subsequently restart the device with the copied application. In addition, it can create new memory partitions on the external FLASH and format it. To this end, we implemented a dynamic partitioning system that allows the use of the external FLASH memory as a WORM (Write Once Read Many) device, without knowing in advance the number and the size of the partitions. To keep track of the first memory location for each application we store in the external FLASH, we use a hash-function returning an identifier (*application ID*) 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. In the external FLASH 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 FLASH and load it. In summary, the `BootLoader` supports the following functionalities: *execute* an application, *format* the external FLASH, *copy* applications from the internal to the external FLASH, *load* applications from the external FLASH. Finally, a portion of the internal FLASH is used to support the communication between `BootLoader` and `DataDissemination`, which can thus exploit the above functionalities at runtime using the `BootLoaderCommunication` interface.

C. Data Dissemination Protocol

Next, we present the data dissemination and error recovery algorithms we implemented in SYNAPSE. Efficient dissemination requires the subdivision of files into so called *transport blocks* of appropriate size, so that they can be processed in the

available RAM. *Transport blocks* are composed of K packets, whose reliable transmission to neighboring nodes is obtained through a Fountain Codes based HARQ protocol. Next, we detail the dissemination protocol in the single-hop case as well as the current implementation of the multi-hop scheme.

Single-hop dissemination: We observe that plain ARQ is inefficient as its throughput quickly decreases as a function of the number of receivers [9]. To overcome this, we adopted a HARQ solution where *transport blocks* are encoded prior to transmission. Differently from ARQ, all we need to know is how many redundancy packets are still needed by each receiver to recover the original data. In addition, the same redundancy packets can correct losses at multiple receivers. Fountain Codes were selected as they allow for lightweight implementations, with advantages in terms of memory requirements and computational needs. They are rateless, i.e., incremental redundancy can be obtained on the fly without needing to know in advance the worst case error probability, the number of receivers, etc. Moreover, they retain the good properties of standard forward error correcting (FEC) codes. The design of FCs is described in Section IV.

Consider a given node i and let $\mathcal{N}_i$ be the set of devices in its communication range. The objective of the single-hop dissemination protocol (`SingleHop` module) is to *reliably* and *efficiently* disseminate the *transport blocks* stored at node i to all interested devices in $\mathcal{N}_i$. Each *transport block* is sent during a so called *dissemination round*. A new round should be initiated only when all nodes in $\mathcal{N}_i$ have received and decoded the current *transmission block*.

At the beginning of a round, node i broadcasts $K + \delta_1$ encoded packets, where $\delta_1 = 4$ is selected to guarantee a recovery probability higher than 0.8 for typical packet error rates, $p \approx 0.05$ and $K = 32$. These packets are followed by a DECODE message. Data packets are used to build a decoding matrix $\mathbf{G}$ (see Section IV). In case a receiver $j \in \mathcal{N}_i$ is still unable to invert $\mathbf{G}$ after receiving the DECODE message, it will ask the transmitter for additional redundancy packets. In this request (NACK) it will indicate the rank of its decoding matrix r_j. The transmitting node i collects incoming NACKs and calculates $\xi = \min_j r_j$. Note that at least $K - \xi$ redundancy packets need to be transmitted to have a full rank $\mathbf{G}$ at all receivers. We chose to transmit $K - \xi + \delta$ packets to provide extra-protection. $\delta = 4$ gives good performance in practical settings as well as over error prone links, i.e., $p \leq 0.3$. Also, δ is kept fixed for all subsequent retransmission requests. Timeouts and a limit on the maximum number of re-transmission cycles are used to avoid deadlocks. In case no NACKs are received after a predefined timeout, node i assumes an implicit ACK from the receivers.

We observe that receiving nodes may have different memory writing times, due to, e.g., different error patterns, battery level, etc. Hence, different nodes will finish writing the *transport block* at different time instants and the transmitter needs to synchronize with the slowest node in order to start the transmission of a new block. This is achieved by electing a *synchronizer* node which, round-by-round, is the slowest node to decode. This node shall provide an explicit acknowledgment to the transmitter upon the completion of each *transmission round*.

Multi-hop dissemination: in its current version, SYNAPSE implements a hop-by-hop data dissemination protocol. In detail, when a node receives the whole file it starts broadcasting advertisement (ADV) messages. ADVs are de-synchronized through random back-offs as multiple nodes may complete the reception of the file at the same time. Upon the reception of an ADV, a node which does not have the data to be disseminated responds with a request (REQ) message; feedback suppression is used to limit the amount of signaling traffic. Thus, the sender starts a new single-hop dissemination phase, intended to all its potential receivers. Network allocation vectors (NAV) are inserted in each ADV/REQ and used to infer the amount of time an overhearing sender should wait before sending its own ADV(s). The details of the ADV/REQ phase are similar to what implemented in [2]. It is observed that the use of FEC allows for smart implementation of pipelining. The focus of the present paper is on the design of codes with good performance as well as their usage within data dissemination schemes in WSNs. Enhanced pipelining techniques exploiting such codes are the objective of our current research and will be integrated in future versions of the system.

IV. Fountain Based Encoding

In section IV-A we discuss the main characteristics of Fountain Codes, why we use them in our framework and which are their main differences with respect to other encoding methods. In section IV-B we detail the Fountain Codes we use in SYNAPSE, discussing the approach we considered for the optimization of the degree distribution at the encoder and the decoding technique we use at the receiving side. In Section IV-C we give some important implementation details.

A. Introduction to Fountain Codes

Digital Fountain Codes were presented by M. Luby in [10] and were the first example of codes realizing the Digital Fountain paradigm of [11]. FCs are near optimal rateless codes designed for erasure channels [6], [12]. Common methods for reliably transmitting packets over these 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 has the advantage of working 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 [9]. As a solution, researchers used HARQ schemes, see, e.g., [1], [9]. 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 for network programming. These, in fact, retain the good performance of previous HARQ schemes [1], [9] 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.
- 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 [1], [9], which use more complex operations among polynomial coefficients in $GF(q)$, with $q > 2$. In fact, fast operations over $GF(q)$ require the use of look-up 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 [13] also perform encoding in $GF(2)$, they are not rateless.

Encoding Procedure: the encoding process is very simple. Its 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 t_n. The input file is subdivided into a number of, say, K packets of b bits each and the following operations are executed:

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

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, 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 IV-B. Finally, rateless codes require the correct reception of $N \geq K$ encoded packets for decoding the original K packets; N depends on the selected distribution $\rho(d)$. For large K, there are encoding distributions requiring a small overhead [10]. The overhead (O) is defined as the extra redundancy needed for recovery, i.e., $O = N - K$.

Decoding Procedure: decoding can be done by inverting a decoding matrix $\mathbf{G}$, which is formed by the received encoding vectors, i.e., solving for $\mathbf{s}$ the system $\mathbf{t} = \mathbf{G}\mathbf{s}$, where $\mathbf{t}$ is the vector containing the received encoded packets, whereas $\mathbf{s}$ contains the K original packets to be retrieved. Very efficient decoding procedures, based on message passing, were proposed for large K [10]; these heuristically solve the above linear system. Our focus in this paper is however different as K in our settings is small. Here, with K we mean the number of packets in a *transport block*, see Section III-C. We remind that, due to the inherent RAM limitations of sensor devices, we cannon work with large K values. Hence, the suboptimal decoding in [10] is not an option in our case due to its poor performance ($O \gg 1$) for small K. On the other hand, we note that optimal decoding amounts to reducing the decoding matrix $\mathbf{G}$ to upper triangular form via Gaussian elimination [12]. 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 overhead at the cost of a reasonable complexity.

B. Optimization of the Degree Distribution $\rho(d)$

For properly designed fountain codes, N should be close to K. Some overhead is unavoidable and depends on the adopted $\rho(d)$. In this section, we present an original algorithm for the optimization of the degree distribution according to given performance objectives. As we show later, our optimization technique is very effective and competitive with state of the art optimizers for Fountain Codes. The optimized degree distributions we present in this section are used within SYNAPSE's error recovery scheme. We optimize our codes for transmission over error-free channels. For full recovery at the receiver(s), all we need is 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, a probability $p > 0$ does not change anything at the receiver side (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 [12].

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 is the number of packets in the input file.
- $p_j, j = 1, 2, \ldots, K$ are the point probabilities defining the degree distribution $\rho(d)$.
- M is the number of samples generated during each iteration of the algorithm.
- $C(i), i = 1, 2, \ldots, M$ is the cost (used to drive the optimization) associated with the i-th sample of the current iteration.
- $N(i), i = 1, 2, \ldots, M$ is the number of encoded packets needed for correct decoding of the original K packets for sample i.
- $n_j(i), i = 1, 2, \ldots, M$; $j = 1, .., K$ is the number of degree j packets generated within the i-th sample.

We use ideas from the theory of genetic algorithms to iteratively obtain, through subsequent refinements, 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:

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

where $|\mathcal{S}|$ is the cardinality of set $\mathcal{S}$. Since this new distribution (it is easy to verify that $\sum_{j=1}^{K} p_j^{new} = 1$) is obtained from samples having small cost, 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 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 the optimization scheme in [14]. In [14] an iterative simulation approach based on importance sampling and gradient search is proposed and used to optimize the degree distribution of LT codes [10], i.e., considering a message passing decoder. As a benchmark to test the effectiveness of our optimization approach, we considered the sparse degree distributions of [14], i.e., $p_j \neq 0$ for $j = 2^\ell$, where $\ell = 0, 1, \ldots, \ell_{max}$, $2^{\ell_{max}} < K$ and $p_j = 0$ otherwise. We initialized each of the non-zero probabilities to the same value, $p_j = 1/\gamma$, such that $\sum_{j=1}^{K} p_j = 1$. We set $C(i) = N(i)$, $\forall\ i$, $M = 1000$, $\alpha = 0.05$ and C_α was updated only when the cardinality of $\mathcal{S}$ was found to be higher than $M/2$, and left unchanged otherwise. We empirically verified that updating C_α only when $\mathcal{S}$ contains a sufficient number of samples leads to better performance. The value $M/2$ was empirically found to give the best performance in our tests. The results of the optimization are shown in Table I; for comparison we also show the values obtained in [14]. As shown in the table, even though our algorithm adopts an empirical approach and no rigorous criteria are defined for its convergence, the performance achieved by our distributions is comparable with that of [14]. Further, it is observed that the results in Table I (and the corresponding distributions) were obtained with at most 20 iterations of $M = 1000$ samples each, i.e., we achieve a gain of orders of magnitude in terms of computational complexity with respect to the optimization technique in [14] (requiring millions of iterations).

TABLE I
OPTIMIZED SPARSE $\rho(d)$ FOR LT CODES (MESSAGE PASSING DECODER).

K →	16	32	64	128
p_1	0.221	0.212	0.161	0.187
p_2	0.457	0.351	0.400	0.339
p_4	0.188	0.288	0.256	0.275
p_8	0.134	0.101	0.101	0.101
p_{16}	-	0.048	0.045	0.046
p_{32}	-	-	0.037	0.031
p_{64}	-	-	-	0.021
$\mathbb{E}[N]$	22.6	43.6	82.7	158.7
Std. $\sigma(N)$	4.4	6.4	9.1	11.4
$\mathbb{E}[N]$ in [14]	22.5	43.6	81.9	159.8
Std. $\sigma(N)$ in [14]	4.2	6.8	7.7	12.1

As discussed in Section IV-A the message passing (LT) decoder [10] 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. The first one, as in the previous example, is the number of packets needed to decode the transmitted file, i.e., $C_1(i) = N(i)$. The second one, $C_2(i)$, is instead defined as the number of XORs between 16-bit words performed at the decoder to recover the original K packets. In detail, $C_2(i)$ is given by the sum of the XORs necessary for the reduction of the received encoding matrix $\mathbf{G}$ in upper triangular form and those needed to recover the original data once $\mathbf{G}$ has been reduced. 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 Eq. (1), that in this case was obtained considering two different values for the α-percentiles for C_1 and C_2; M was set to 50000 and the initial distribution was $p_j = 1/K$, $j = 1, 2, \ldots, K$. In order to adhere to our experimental settings, we considered packets of 25 bytes that for $K = 32$ correspond to transport blocks of 800 bytes. XORs operate on 16 bit words, as for the TI MSP430 micro-controller of our sensor nodes. Optimizations were carried out for $K \in \{32, 48, 64, 128\}$. Due to space constraints, here we only present results for $K = 32$, which was also considered for the results in Section V. The selection of the parameters α_1 and α_2 to use within the algorithm is not trivial; by tuning these two coefficients it is in fact possible to obtain degree distributions with different properties in terms of overhead ($O = N - K$) and decoding cost (number of XORs to obtain the K original packets). Note that a lower decoding cost will result in a higher overhead and vice versa. We ran an extensive optimization campaign varying these parameters. The distributions leading to minimum overhead and minimum decoding cost were obtained setting (α_1, α_2) to $(0.05, 1)$ and $(1, 0.05)$, respectively. For $K = 32$ we obtained the distributions shown in Figs. 2 and 3. Besides these two distributions, we also consider the uniform distribution as it is known to give asymptotically optimal performance in terms of overhead [6] (even though it has unsatisfactory cost performance). For our decoder the uniform distribution achieves an average overhead of $\mathbb{E}[C_1] = 34.09 \pm 0.03$ packets and an

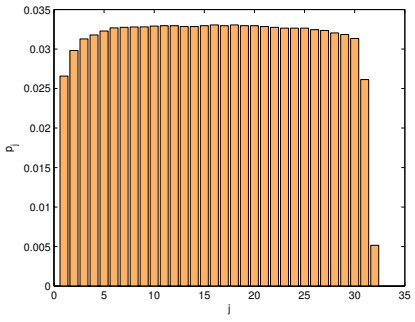

Fig. 2. $\rho(d)$: optimal encoding distribution in terms of transmission overhead.

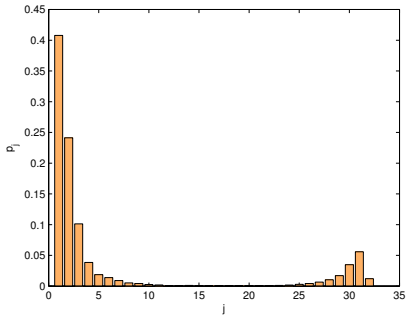

Fig. 3. $\rho(d)$: optimal encoding distribution in terms of overall decoding cost.

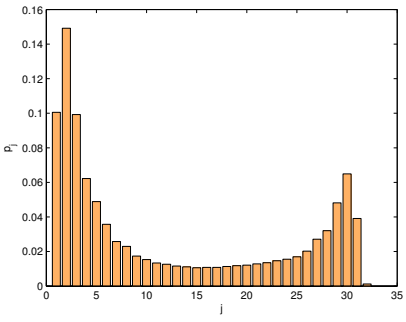

Fig. 4. $\rho(d)$: encoding distribution obtaining a good tradeoff between overhead and cost.

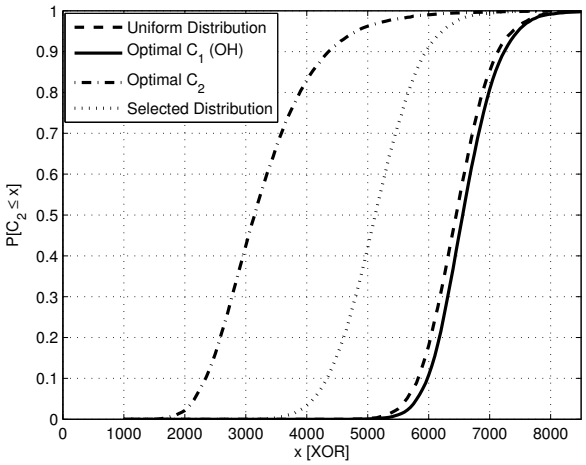

Fig. 5. Empirically measured cumulative distribution of the computational cost at the decoder for different degree distributions. The cost is measured as the number of XORs between 16-bit words.

TABLE II
OPTIMIZED $\rho(d)$ FOR A GAUSSIAN ELIMINATION DECODER, $K = 32$.

$j = 1 \rightarrow 16$	p_j	$j = 17 \rightarrow 32$	p_j
1	0.1005	17	0.0108
2	0.1493	18	0.0113
3	0.0993	19	0.0118
4	0.0622	20	0.0121
5	0.0489	21	0.0128
6	0.0357	22	0.0135
7	0.0258	23	0.0147
8	0.0230	24	0.0156
9	0.0174	25	0.0169
10	0.0154	26	0.0202
11	0.0134	27	0.0271
12	0.0126	28	0.0321
13	0.0116	29	0.0482
14	0.0111	30	0.0650
15	0.0106	31	0.0391
16	0.0108	32	0.0012

average decoding cost of $\mathbb{E}[C_2] = 6481 \pm 10$ XORs; 95% confidence intervals were obtained considering 10000 samples of the decoding process. In Fig. 2 we plot our optimal distribution in terms of overhead, having $\mathbb{E}[C_1] = 33.65 \pm 0.03$ packets and a slightly higher decoding cost of $\mathbb{E}[C_2] = 6586 \pm 9.8$ XORs. For small K it performs better than the uniform distribution. The best performance in terms of decoding cost is achieved with the distribution in Fig. 3 having $\mathbb{E}[C_2] = 3249 \pm 16$ XORs, i.e., its decoding cost is more than 50% smaller than that achievable with the uniform distribution (see Fig. 5). However, this last distribution is not interesting in practice as its average overhead is unacceptably high, i.e., $\mathbb{E}[C_1] = 50.7 \pm 0.2$ packets.

As one might expect, suitable tradeoffs between overhead and decoding cost can be obtained through a judicious choice of the pair (α_1, α_2). For the selection of these parameters we have done an exhaustive search in the region $\{(\alpha_1, \alpha_2) : 0 \leq \alpha_1 \leq 1, 0 \leq \alpha_2 \leq 1\}$, from which we selected the distribution in Fig. 4, obtained considering $\alpha_1 = 0.05$ and $\alpha_2 = 0.075$. For this distribution we have $\mathbb{E}[C_1] = 34.26 \pm 0.04$ and $\mathbb{E}[C_2] = 5142 \pm 12$, thus we reduce the decoding cost of more than 20% with respect to the uniform case (see Fig. 5) whilst maintaining almost the same overhead. The optimized degree distribution for this case is shown in Table II. We observe that these results substantially improve the results shown in Table I: this is mainly due to the higher performance of Gaussian elimination with respect to message passing decoding. In addition, in these last optimizations we did not restrict ourselves to a specific class of distributions, as we instead did for the results in Table I. We finally observe that higher gains can be obtained considering larger values for K. As an example, with $K = 128$ a cost reduction of up to 40% can be achieved with respect to the uniform distribution, whilst maintaining almost the same overhead. This K, however, hardly fits our memory requirements.

C. 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 packets identifiers. An initial seed is communicated at the beginning of *transmission rounds*, whereas the seeds used for the subsequent packets are incrementally obtained from their sequence numbers. In this way, no overhead is introduced for the transmission of encoding vectors. In addition, the choice of the pseudo random generator deserves particular attention. In SYNAPSE we adopt a generator based on Linear Feedback

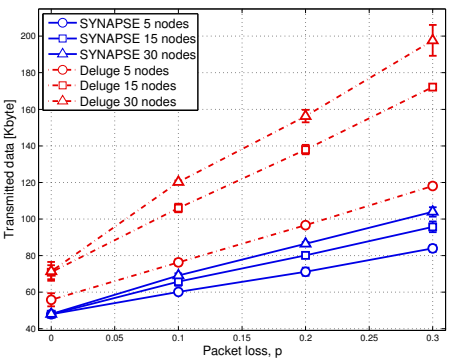

Fig. 6. Data traffic vs. p.

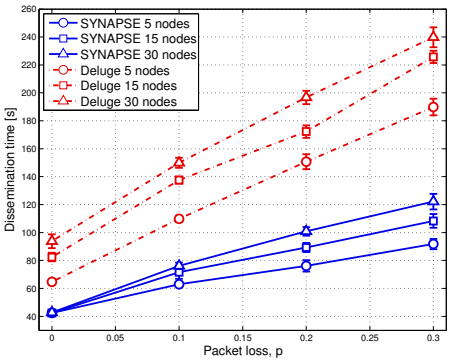

Fig. 7. Data dissemination time vs. p.

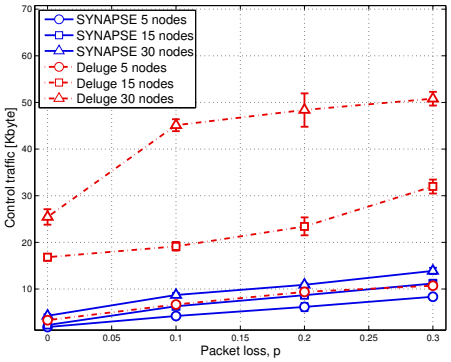

Fig. 8. Signaling traffic vs. p.

Shift Registers (LFSR) [15] working with registers of 16 bits. This method, which is optimized for the TI MSP430 micro-controller of our sensor nodes, is very fast. Decoding a block of $K = 32$ packets (800 bytes) with LFSR takes about 462 ms. For comparison, the same operation with a more accurate Linear Congruential random Generator takes about 660 ms. A drawback of LFSR is that a few random seeds exist which provide unsatisfactory performance. There is, however, a large number of seeds for which LSFR performs properly.

V. Experimental Results

The experimental results that we show in this section were obtained in the SignetLab testbed deployed in the Department of Information Engineering of the University of Padova [16]. The hardware platform consists of Tmote Sky sensor nodes, featuring an IEEE 802.15.4 2420 Chipcon wireless transceiver working at 2.4 GHz and allowing a maximum data rate of 250 Kbps. These sensors have a TI MSP430 micro-controller with 10 Kbytes of RAM and 48 Kbytes of internal FLASH. These nodes are also equipped with an external FLASH memory of 1 Mbyte. SYNAPSE was developed using the nesC programming language in TinyOS v2.x [17].

In what follows, the performance of SYNAPSE is compared to that of Deluge [2]. In our comparison between SYNAPSE and Deluge, exactly the same amount of data was transmitted by the two dissemination protocols in all experiments. Note that, as mentioned in Section II, the optimizations introduced in Stream [4], where the actual amount of data to disseminate is reduced by pre-installing the re-programming software in all sensor nodes, can be used in SYNAPSE as well. The performance enhancement of optimized SYNAPSE compared to Stream would be similar to that of SYNAPSE compared to standard Deluge.

We ran a series of tests to assess the performance of SYNAPSE in a single hop environment with one sender and a variable number of receiving nodes. In order to have full control of the packet error probability over the wireless links, the receivers were positioned sufficiently close to the transmitter so as to have a negligible packet error probability due to channel impairments, and we emulated channel errors by discarding the received packets through a software defined probability p. In the following plots, vertical bars are used to represent 95% confidence intervals.

As a first result, Fig. 6 shows the total number of data bytes transmitted by all nodes to successfully disseminate a program of 27100 bytes to all receivers. The results for 5, 15 and 30 receivers are plotted as a function of $p \in [0, 0.3]$. Considering the experiments with 5 receivers, we note that there is an initial gap for $p = 0$, which is due to the publishing procedure implemented in Deluge. In detail, as soon as any node receives a correct page, it starts publishing this information through ADV messages. This is necessary to achieve spatial multiplexing [2], which consists of a distributed and very efficient technique for multi-hop dissemination and ARQ. However, as a drawback of this mechanism data packets may collide; this effect is more pronounced for an increasing network density (see curves for 15 and 30 nodes and $p = 0$).

It is observed that, with this phenomenon alone, Deluge's and SYNAPSE's curves would be parallel. This, however, does not occur but the performance gap between the two protocols increases with p. This is mainly due to the higher efficiency of SYNAPSE's HARQ technique. Note that the authors of Deluge [2] decided against using error correcting codes as they found that, for realistic network settings, this did not offer good results, which is expected if traditional fixed-rate FEC is used. On the other hand, SYNAPSE uses a more sophisticated FEC technique which is efficient in these cases, as the rateless codes we use do not need to know the error probability experienced by the receivers in advance, but rather adapt to the actual channel conditions. Also, the code is efficient for a large number of receivers as additional redundancy can be generated on the fly until full recovery.

In Fig. 7, we show the reprogramming time as a function of p for the same network configurations as above. We observe a trend similar to that in Fig. 6, as well as substantial improvements in terms of reprogramming time. Note also that the gap between Deluge's curves is mainly due to the higher number of collisions of ADVs, REQs and DATA packets experienced for an increasing number of nodes. In SYNAPSE, instead, this gap is mainly due to the collisions occurring over the feedback channel, i.e., among NACKs. These collisions slow down the dissemination and are more likely to occur when the node density increases. A feedback suppression mechanism is

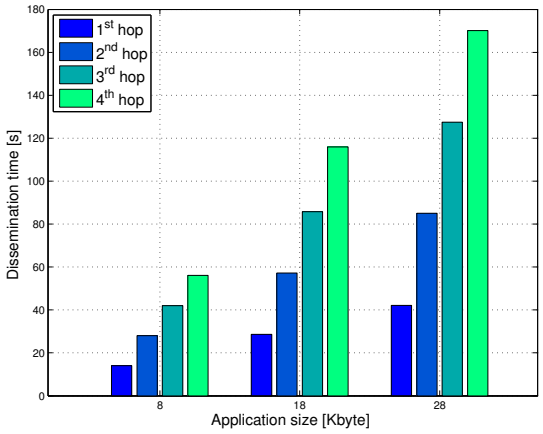

Fig. 9. SYNAPSE's dissemination time for a multihop network with 4 hops and 30 nodes.

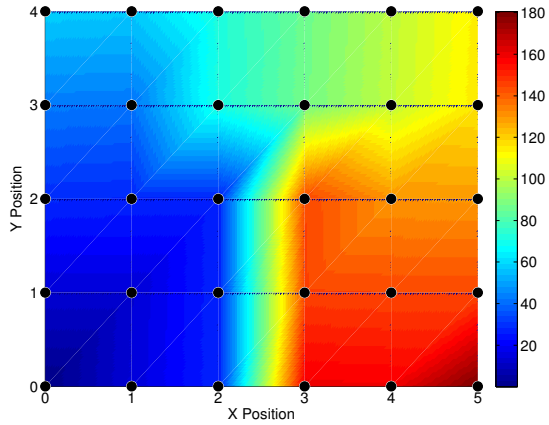

Fig. 10. SYNAPSE: snapshot of a single dissemination for a grid network of 30 nodes. Different colors represent different dissemination times (in seconds).

however used in SYNAPSE to mitigate this problem.

We continue our discussion with Fig. 8, where we show the signaling traffic sent by the two dissemination protocols as a function of p. Interestingly, Deluge performs very close to SYNAPSE at low densities (5 nodes), whereas for an increasing number of receivers its performance is considerably impacted by the transmission of control packets. This is not due to the Trickle algorithm [2] that Deluge uses for the suppression of ADVs, but rather to the fact that all receivers have to send their error bit-vectors (REQs). This is not necessary in our case as redundancy packets are effectively used to correct different losses at multiple receivers. Hence we only need to receive at least one NACK requiring a sufficient number of packets, rather than multiple specific retransmission requests.

As a sample result for SYNAPSE's dissemination time in multi-hop scenarios, in Fig. 9 we show the reprogramming time for the first 4 hops of a grid network. In Fig 10 we show a snapshot of the reprogramming times for the same network. As expected from the hop-by-hop nature of the implemented multihop mechanism, the dissemination time is linear with the number of hops. This shows that SYNAPSE is also effective in multi-hop environments. Improvements of SYNAPSE in these scenarios are the objective of our future research.

VI. Conclusions

In this paper we presented SYNAPSE, a system for reprogramming WSNs exploiting rateless Fountain Codes. We first reviewed the advantages offered by these codes. We subsequently designed, through a novel genetic optimization approach, an encoding distribution which is tailored to the specific needs of WSNs. This distribution is used at the heart of SYNAPSE's dissemination and error recovery mechanisms. Finally, we tested our TinyOS implementation of the protocol on the Tmote Sky sensor platform. Experimental results, obtained for single as well as multi-hop environments, confirm the effectiveness of our approach in disseminating data to a large number of receivers, especially in the presence of channel impairments. In conclusion, rateless codes proved to be a viable and very promising practical method for disseminating data in WSNs. In the final version of the paper we will provide a link to the open source code distribution of SYNAPSE.

References

[1] J. Nonnenmacher, E. W. Biersack, and D. Towsley, "Parity-based Loss Recovery for Reliable Multicast Transmission," *IEEE/ACM Trans. on Networking*, vol. 6, no. 4, pp. 349–361, 1998.

[2] J. W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," in *ACM SenSys*, Baltimore, Maryland, USA, Nov. 2004.

[3] S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *IEEE ICDCS*, Columbus, Ohio, USA, Jun. 2005.

[4] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," in *IEEE INFOCOM*, Anchorage, Alaska, USA, May 2007.

[5] 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. on Sensor Networks*, 2008, accepted for publication.

[6] D.J.C. MacKay, "Fountain Codes," *IEE Proceedings – Communications*, vol. 152, no. 6, pp. 1062–1068, 2005.

[7] J. Jeong, S. Kim, and A. Broad, "Network Reprogramming," Berkeley, California, USA, Aug. 2003. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/

[8] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," Los Angeles, California, USA, 2003, CENS Technical Report no. 30.

[9] M. Rossi, M. Zorzi, and F. H. Fitzek, "Link Layer Algorithms for Efficient Multicast Service Provisioning in 3G Cellular Systems," in *IEEE Globecom*, Dallas, Texas, US, Nov. 2004.

[10] M. Luby, "LT Codes," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, Vancouver, B.C., Canada, Nov. 2002.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *ACM SIGCOMM*, Vancouver, B.C., Canada, Sep. 1998.

[12] D.J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[13] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical loss-resilient codes," in *29-th annual ACM symposium on Theory of computing*, El Paso, Texas, US, May 1997.

[14] E. Hyytiä, T. Tirronen and J. Virtamo, "Optimizing the Degree Distribution of LT Codes with an Importance Sampling Approach," in *RESIM 2006, 6-th International Workshop on Rare Event Simulation*, Bamberg, Germany, Oct. 2006.

[15] D. E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997.

[16] R. Crepaldi, S. Friso, A. F. Harris III, M. Mastrogiovanni, C. Petrioli, M. Rossi, A. Zanella, and M. Zorzi, "The Design, Deployment, and Analysis of SignetLab: A Sensor Network Testbed and Interactive Management Tool," in *IEEE Tridentcom*, Orlando, Florida, US, May 2007.

[17] "TinyOS: an open source OS for the networked sensor regime." [Online]. Available: http://www.tinyos.net