TECHNISCHE
UNIVERSITÄT
DARMSTADT

RANDOM NETWORK CODING BASED BROADCAST

JAN STURM

Lab Report

March 15, 2018

Secure Mobile Networking Lab
Department of Computer Science

SECURE MOBILE NETWORKING

Random Network Coding based Broadcast
Lab Report
SEEMOO-MSC-0000

Submitted by Jan Sturm
Date of submission: March 15, 2018

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Dr.-Ing. Dingwen Yuan

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

The focus of this lab is to implement a simulation of Random Network Coding (RNC) based broadcast for wireless sensor networks. The open source operating system Contiki, together with the included network simulator Cooja, will be used to simulate sensor nodes which perform RNC. The focus lies on a reliable RNC broadcast mechanism with low latency and low power usage, resp. efficient algorithms and efficient computations in the chosen Galois fields. Furthermore we implement a simple flooding scheme which provides a fair comparison between both schemes. The simulations show that RNC can outperform a naive flooding scheme with regard to power consumption and latency if we chose high enough batch-sizes and big enough Galois fields.

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Wireless Sensor Networks are widely used for various purposes, e.g. industrial monitoring, area monitoring or medical applications. They are built with sensor nodes, devices which are usually very basic in terms of their interfaces and their components. These nodes consist of a small micro-controller, limited memory, a radio transceiver, a battery power supply and the sensors. They process data, gather data and communicate with other sensor nodes in the network. When sensor nodes are ready for deployment in the environment, the firmware will most likely contain environment-dependent bugs. These bugs need to be debugged and eliminated by frequent software updates. This is one scenario where we need a reliable, fast and energy efficient broadcast mechanism in order to distribute packets to the nodes. Consider a network consisting of a source, relays and sinks. A source node continuously broadcasts a batch of packets (e.g. firmware updates) throughout the network, while the intermediate relays try to forward the packets to the sinks. In the end each sink should receive every packet the source has sent out. A naive approach would be to flood the packets through the network. Each relay broadcasts the received packet again until it reaches the sink nodes. Although it is simple to implement this naive approach is not very energy efficient, especially for wireless sensor networks where power is one of the most critical resources and packet transmission is a very energy-consuming action for sensors. With Random Network Coding we can reduce the traffic and increase the reliability in those single-source broadcast scenarios.[4, 7, 13]

1.2 BACKGROUND

1.2.1 *Random Network Coding*

The core idea of network coding is to allow the mixing of data at intermediate network nodes. While the source node continuously broadcasts a batch of packets throughout the network, each intermediate network node will help to spread the packets they receive by sending out one packet containing a linear combination of those packets. The randomness in *Random* Network Coding is due to random linear combinations, thus random coefficients. Each relay node constructs

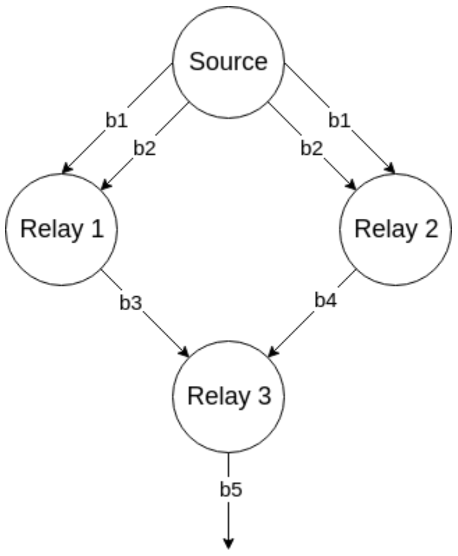


Figure 1: simple RNC scenario with $K = 2$

a linear combination of K packets $p_1, \dots, p_K$ by choosing K random coefficients $a_1, \dots, a_K$ and subsequently compute

$$b = a_1 \cdot p_1 + a_2 \cdot p_2 + \dots + a_K \cdot p_K.$$

After that it will broadcast the packet b together with all coefficients $a_1, \dots, a_K$. The receiving nodes will perform Gaussian elimination (see Section 1.2.2) to decode the original packets $p_1, \dots, p_K$. Furthermore all arithmetic operations will be calculated in a Galois Field (see Section 1.2.3), which means that combined packets will have the same size as any of the individual packets. Figure 1 demonstrates a simple RNC example of encoding packets which will emphasize this property. We denote a combined packet b_i by

$$b_i : \left[\texttt{header}_i \;\middle|\; \texttt{pl}_i\right] = \left[[a_1, \dots, a_K]_i \;\middle|\; \texttt{pl}_i\right] \quad \texttt{, with pl as the payload}$$

Consider $K = 2$ and two uncoded packets b_1, b_2 sent by the source:

$$b_1 : [\ 1,0 \mid \texttt{pl}_1\ ], \quad b_2 : [\ 0,1 \mid \texttt{pl}_2\ ]$$

The three packets b_3, b_4, b_5 will be constructed by relays as follows:

$$\begin{aligned} b_3 : a_1 \cdot b_1 + a_2 \cdot b_2 &= [\ a_1, a_2 \mid a_1 \cdot \texttt{pl}_1 + a_2 \cdot \texttt{pl}_2\ ] \\ b_4 : a_3 \cdot b_1 + a_4 \cdot b_2 &= [\ a_3, a_4 \mid a_3 \cdot \texttt{pl}_1 + a_4 \cdot \texttt{pl}_2\ ] \end{aligned}$$

$$\begin{aligned} b_5 :\ & a_5 \cdot b_3 + a_6 \cdot b_4 \\ &= [(a_5 \cdot a_1 + a_6 \cdot a_3), (a_5 \cdot a_2 + a_6 \cdot a_4) \mid a_5 \cdot \texttt{pl}_3 + a_6 \cdot \texttt{pl}_4\ ] \\ &= [(a_5 \cdot a_1 + a_6 \cdot a_3), (a_5 \cdot a_2 + a_6 \cdot a_4) \mid \\ &\quad (a_1 \cdot a_5 + a_3 \cdot a_6) \cdot \texttt{pl}_1 + (a_2 \cdot a_5 + a_4 \cdot a_6) \cdot \texttt{pl}_2\ ] \\ &= [\ \tilde{a}_5, \tilde{a}_6 \mid \tilde{a}_5 \cdot \texttt{pl}_1 + \tilde{a}_6 \cdot \texttt{pl}_2\ ] \end{aligned}$$

The coefficient vectors $[a_1, a_2], [a_3, a_4]$ and $[\tilde{a}_5, \tilde{a}_6]$ are ideally pairwise linear independent.

However RNC has some challenges to solve. We need to find an efficient way to calculate the Gaussian elimination procedure. Furthermore there is a limit of how much packets we can send in one batch since the coefficient matrix grows exponentially with every additional packet and sensor nodes have only limited memory to buffer received packets. Another big issue is to deal with linear dependency among coefficient vectors. The larger our Galois field the more unlikely are the coefficients to be linear dependent, but the computation costs might increase.[10]

1.2.2 *Gaussian Elimination*

Definition 1.2.1 (Linear dependency) *Vectors* $a_1, \ldots, a_n$ *of the same dimension are called linearly independent, if*

$$x_1 \cdot a_1 + \cdots + x_n \cdot a_n = 0$$

implies $x_1 = \cdots = x_n = 0$. *Otherwise they are called linearly dependent.*

Definition 1.2.2 (Rank) *The rank of a matrix* A *is the maximal number of linearly independent row vectors of* A.

We construct a coefficient matrix A where each row i corresponds to the coefficient vector of a received packet i. Each payload $\mathtt{pl}_i$ will be placed in row i of a column vector b. We obtain a system of linear equations $A \cdot x = b$, where x are the decoded packets that were sent out by the source in a single batch.

$$A \cdot x = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1K} \\ a_{21} & a_{22} & \ldots & a_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K1} & a_{K2} & \ldots & a_{KK} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix} = b$$

The step by step process of solving this system of linear equations by eliminating the unknowns in the system is called Gaussian elimination. We perform elementary row operations on the augmented matrix $A|b$ to obtain an upper triangular matrix $\tilde{A}$, i.e. a matrix where the lower left-hand corner is filled with zeros:

$$\tilde{A} \cdot x = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1,K-1} & a_{1K} \\ 0 & \tilde{a}_{22} & \ldots & \tilde{a}_{2,K-1} & \tilde{a}_{2K} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & \tilde{a}_{K-1,K-1} & \tilde{a}_{K-1,K} \\ 0 & 0 & \ldots & 0 & \tilde{a}_{KK} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{K-1} \\ x_K \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b}_2 \\ \vdots \\ \tilde{b}_{K-1} \\ \tilde{b}_K \end{bmatrix} = \tilde{b}$$

Elementary row operations are

- swap two rows
- multiply a row by a non-zero number
- add a multiple of a row to another row

If $\texttt{rank}(A) = K$ there exists exactly one solution to the linear system, so in order to fully decode a batch, we need K linearly independent packets, i.e. all coefficients must be linearly independent. With back substitution we can decode the packets $x_1, \dots, x_K$. [6]

1.2.3 *Galois Fields*

A field is a set of elements for which addition, multiplication, subtraction and division performed with its elements result in another element of the same set. The number of elements of a field is called the order of that field. A field with a finite number of elements is called a finite field, or Galois field GF. For every prime number p and every positive integer m, there exist finite fields of order p^m, which are denoted GF(p^m). GF(2) for example consists of the elements 0 and 1, where addition is exclusive OR (XOR) and multiplication is AND. Galois fields of practical interest are of the form GF(2^m), which are extensions of the binary field GF(2), with m a positive integer number. Elements of GF(2^m) can be represented either as polynomials of degree strictly less than m over GF(2) or as binary numbers. Example for GF(2^8):

HEXADECIMAL: $0xa3$

BINARY: 10100011

POLYNOMIAL: $x^7 + x^5 + x + 1$

Operations are performed modulo an irreducible polynomial of degree m over GF(2). Addition and subtraction is done with an XOR operation on the binary representation, while multiplication is performed by a polynomial multiplication followed by division using the irreducible polynomial as the divisor. The multiplicative inverse of an non-zero element a can be calculated with the extended Euclidean algorithm or by the calculation of a^{2^m-2}. For the latter, we use the property that $a^{2^m-1} = 1$. In order to optimize the multiplication and multiplicative inverse operations, we can leverage table look-ups. First we need to find a generator g of GF(2^m). That is a primitive ($2^m - 1$)th root of unity in GF(2^m), which means that each non-zero element of GF(2^m) can be written as g^i for an integer i. After finding such a generator g, we can rewrite the multiplication of two elements a, b as

$$a \cdot b = g^{\log_g(a \cdot b)} = g^{(\log_g(a) + \log_g(b)) \mod |g|},$$

where $|g|$ denotes the order of the generator, i.e. the number of non-zero elements of GF(2^m). We need to precompute two look-up tables, one for the $\log_g(x)$ function and one for the g^x function, each holding 2^m values. Effectively there are three table look-ups for one multiplication.
With the same precomputed tables we will calculate the multiplicative inverse of a as

$$a^{-1} = g^{\log_g(a^{-1})} = g^{-\log_g(a)} = g^{|g|-\log_g(a)}.$$

Two table look-ups are necessary. The generation of the tables will be explained in Subsection 2.3.1.[3, 5, 14]

1.3 CONTIKI & COOJA

In the course of this lab we will use the Contiki OS and Cooja as the development platform. Contiki is a powerful toolbox for building complex wireless systems in the programming language C. It is an open source operating system which focuses on the Internet of Things and thus provides powerful low-energy Internet communication. It is designed to operate in extremely low-power systems, that may need to run for years on batteries. Due to the ContikiMAC radio duty cycling mechanism sensors can go to sleep while not transmitting. It is the default mechanism that provides a very good power efficiency, where the transmitted packet must be repeated until the receiver turns the radio on in order to detect the transmission. As we can see in Figure 2, the traffic will be increased but the radio is only on when a transmission occurs.

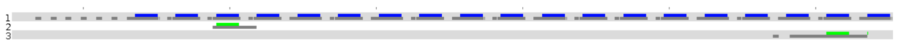


Figure 2: dark grey lines: radio on; blue lines: transmitting; green lines: receiving

Furthermore Contiki provides an own wireless networking stack called Rime. It supports a set of lightweight communication primitives, ranging from simple operations such as a broadcast to more complex mechanisms. The whole Contiki toolbox is combined as *Instant Contiki* in a virtual machine image, making it easy to develop and compile applications for any of the available Contiki platforms.

Instant Contiki includes Cooja, the Contiki network simulator. It provides a simulation environment, which makes it feasible to develop and debug software for fully emulated hardware devices in large wireless sensor networks.[2]

1.4 RELATED WORK

In [7] a team from the University of Illinois propose AdapCode, a reliable data dissemination protocol using adaptive network coding. Their goal is to reduce the broadcast traffic for wireless sensor networks in the process of code updates, e.g. for debugging purposes. The source keeps on sending broadcast messages, while each intermediate node generates K coefficients and computes the linear combination of K packets. Their implementation does not allow nodes to combine and transmit packets that they have not been able to decode yet, although this will reduce the propagation delay. Their approach is to adaptively change the coding scheme according to the link quality, which is that nodes dynamically decide K based on how many neighbors they have. For 100% reliability the nodes will send out Negative-ACKs (NACKs) in order to retrieve missing data in case they received less than K packets. They also choose the coefficients from 0 to $p - 1$, with p prime, rather than from a Galois Field. This might result in an overhead caused by a possible overflow when performing linear combination, although the authors show that the overhead is acceptably small. Furthermore AdapCode is compared against Deluge [8], the most widely used code dissemination protocol. They observe that AdapCode outperforms Deluge in terms of traffic throughput, load balancing and latency.

2 CONTRIBUTION

2.1 PROJECT ORGANIZATION

The lab schedule was divided into three phases. Throughout those phases, meetings were held with the supervisor to discuss possible optimizations and questions.

1. alpha phase
   - efficient arithmetic in GF
   - efficient implementation of Gaussian elimination in GF
   - random linear combination of packets
   - simple Cooja simulation
   - unreliable broadcast
   - decoding of batches at sinks
2. beta phase
   - address linear dependency among coefficients and thus ensure full reliability
   - add more functionality to Cooja simulation
3. final phase
   - find out suitable parametersets
   - implement simple flooding scheme and compare performance

2.2 DESIGN

This section describes our design of the Random Network Coding based broadcast scheme. Our scheme deals with linear dependency among coefficient vectors to ensure 100% reliability and the main focus lies on the reliability- and throughput performance. Algorithm 1 gives a rough overview over our scheme.

2.2.1 *Protocol Overview*

In our network scenario, we can configure a sensor node either as source, relay, sink or relay and sink combined. There is one single source in the system, which will keep broadcasting packets of data

Algorithm 1 Random Network Coding based Broadcast

```
1: initialization
2: mode = source || sink || relay || relay+sink
3: coeffMatrix ← $K \times K$ matrix
4: packetQueue ← $(K \times M)$byte buffer for the encoded packets
5: rec ← $(K \times M)$byte buffer for the recovered packets
6: timer_nack ← $T$
7: while packet distribution in $batch_i$ is going on do
8:     if a RNC-packet or NACK-reply received then
9:         if mode $\neq$ source AND $batch_i$ not recovered then
10:            timer_nack ← $T$
11:            rank_old ← rank(coeffMatrix)
12:            store RNC-packet in coeffMatrix and packetQueue
13:            rank ← Gaussian(coeffMatrix, rec, packetQueue)
14:            if mode == relay OR mode == relay+sink then
15:                if rand_old < rank then
16:                    broadcast new RNC-packet after random period
17:            if rank(coeffMatrix) == $K$ then
18:                stop timer_nack
19:    if a NACK received AND mode $\neq$ sink then
20:        packets ← the missing packets
21:        if any of packets is already received then
22:            wait for a random period
23:            if no NACK-reply heard during the period then
24:                generate and broadcast new rnc packet
25:    if timer_nack timeouts then
26:        send a NACK
27:        timer_nack ← $2 \cdot$ timer_nack
```

Algorithm 2 Gaussian(A,x,b)

```
1: run Gaussian elimination on the augmented matrix $A|b$
2: rank ← rank(A)
3: if mode == sink OR mode == relay+sink then
4:     if rank == $K$ then
5:         solve LSE $A \cdot x = b$ for $x$ with back substitution
6: return rank
```

in a certain time interval. K of these packets form a batch with a certain batch-id. That means, our network coding scheme will combine up to K packets with the same batch-id. It is also possible to combine less than K packets; even only one (uncoded). After the source sent out all packets in a batch, it will timeout for a period T to ensure that each sink received and decoded the payload before a new packet with a higher batch-id arrives. A node marked as relay will help spread packets they receive. It will use Random Network Coding to improve the throughput and reliability of the link, but will not actually decode the payload. The sink nodes in contrast will decode the payload with Gaussian Elimination, but will not participate in packet distribution. A node marked as sink and relay will obviously combine the functionalities of a sink and a relay. As shown in Section 1.2.1, a packet contains the actual data payload, the respective coefficient vector and furthermore the batch-id plus the type of the packet. We differentiate between three message types: a standard RNC packet, a Negative-ACK (NACK) and a NACK-reply. Without NACKs, a sink receiving $K-1$ packets and a sink receiving no packets at all are similar problematic, since neither of them can decode any of the original K packets. During the data distribution process, every node but the source keeps a countdown timer. Initially this timer is set to T seconds. If a node's timer fires, it will send out a NACK to the local broadcast address. A NACK message contains the batch-id as well as an indicator which packets are still missing. A key problem in dealing with NACKs is to determine which node should respond to the NACK. If many nodes respond simultaneously they cause unnecessary transmissions and channel congestion. We use the same approach as AdapCode used in their design. When a node receives a NACK message, it first checks whether it has any of the requested information available. If so, the node will delay for a random period of time to see if any of its neighbors is replying to this NACK. If no NACK-reply is heard before the timeout, this node will respond to the NACK by sending a linear combination of all its packets received so far. Further, we adopt AdapCode's "lazy NACK" mechanism to reduce the number of NACKs. When a node sends out a NACK, it doubles the value of its countdown timer. The value of the timer will be restored to T once the node receives a new packet. Unlike the design decision in AdapCode, we will allow relays to combine and transmit packets that they have not yet been able to decode. As soon as a relay receives a new linearly independent packet, it will generate a random linear combination of the packets received so far and broadcasts it. Although it increases the bandwidth usage compared to AdapCode, it will reduce the propagation delay.

The computations will be performed in $GF(2^m)$. Our implementation supports GF(256), GF(16) and GF(2).

2.2.2 *Optimizations*

In the first phase of the lab we used the inverse matrix A^{-1} to solve the LSE $A \cdot x = b$ for debugging purposes. However this approach is very inefficient, since we need a full-rank coefficient matrix in order to be invertible. Gaussian elimination in contrast is a iterative algorithm. As soon as we receive a new packet, we can run Gaussian elimination on the augmented coefficient matrix and thus built already parts of the upper triangular coefficient matrix. When the next packet arrives we have to do less computation since many entries in the matrix are already zero. With this approach we don't have to perform one big computation once we have a full rank matrix, but smaller computations every time we receive a packet. Another advantage is, that we can determine in-time whether a received packet provides new information.

Furthermore we adapt the highly efficient GF computations for GF(256) and GF(16) from Section 1.2.3 where we precompute two look-up tables for the $\log_g(x)$ and the g^x function. Each table holds 2^m values. All values of the three different GFs can be represented by one byte. For GF(256) we need 2×256 bytes of ROM to hold the values, which is acceptable even for sensor nodes with sparse memory. Obviously we could also use 256×256 bytes of ROM to implement multiplication with one look-up, but this would exceed the memory of the sensor.

Another approach which was implemented is that a node queries the broadcast channel before sending out a RNC-packet. If it sees that another packet is transmitted which covers its own packet it will not send it out, e.g. when it sees that the first three coefficients are set, it will not send out its own packet with only the first two coefficients set. However this approach was discarded for the final implementation since too much useful information was discarded. If a packet is covered by another packet, it could still contain helpful linearly independent data.

2.3 IMPLEMENTATION

The project is distributed over the following directories:

```
~/
└─ contiki
   ├─ core
   │  └─ net
   │     └─ seemoo-lab
   │        ├─ rnc
   │        └─ flooding
   └─ examples
      └─ sky
         └─ flooding
```

```
|
|         └── rnc
└── simulations
```

The files in ~/contiki/core/net/seemoo-lab/ contain the implementation of the actual schemes, while the files in ~/contiki/examples/sky/ provide the respective testbenches. In the following we will mainly focus on the RNC implementation since the additional flooding implementation has a similar structure. But differences will be elucidated to provide a fair comparison.

2.3.1 *RNC*

The RNC implementation is split into several different files which we will examine one after another:

```
~/contiki/core/net/seemoo-lab/rnc/
  ├── gauss.c
  ├── gauss.h
  ├── gf.c
  ├── gf.h
  ├── gf_percomp.h
  ├── gf_precomp_calc.py
  ├── params.h
  ├── rnc.c
  ├── rnc.h
  ├── util.c
  └── util.h
```

params.h

This is the only file the user has to interact with, if he wants to change the parameters for the simulation. We can set the mode of each node by its ID, which Galois Field should be used, the batch size, the packet size and timeout values. The user can also set that debug or demo messages should be printed on the console. Note that console outputs will extremely intervene with the desired behaviour of the scheme, since timeouts will fire to early. For the performance evaluation we will use the powertrace tool which comes with Contiki. In order to see the powertrace printout, `POWERTRACE` must be defined. Furthermore LEDs can be turned on if the nodes finished decoding a batch.

gf.c / gf.h / gf_precomp.h

All GF related operations reside in `gf.c`. A GF element is stored in an *uint8_t* datatype which represents one byte. We can add two elements (`gf_add`), multiply two elements (`gf_mul`), calculate the inverse (`gf_inv`) or choose random elements (`gf_choose_random`). For GF(256) and GF(16), multiplication and inversion is done with the precomputed look-up tables in `gf_precomp.h`. We will perform the division of two elements by calculation of the inverse of one element and then a multiplication of this inverse times the other element. The same operations can be applied on vectors of GF elements. For the Gaussian elimination algorithm we need the dot product of two vectors (`gf_vec_dot_vec`) and the product of a row vector with a matrix (`gf_vec_dot_matrix`).

gf_precomp_calc.py

This python file automatically generates the header `gf_precomp.h` which holds the look-up tables. We will denote the table for the $\log_g(x)$ function as `precomp_log` and the table for the g^x function as `precomp_antilog`. Listing 1 shows how both tables will be filled.

```
g = generator of GF
tmp = 0x01
log = [0]*GF
antilog = [0]*GF

for i in range(GF-1):
    precomp_log[tmp] = i
    precomp_antilog[i] = tmp
    tmp = gf_mul(tmp,g)
```

Listing 1: generation of look-up tables

For the actual non-precomputed `gf_mul` operation we use the code from [12] which uses the so called Russian Peasant Algorithm. For more information on this algorithm see [11].

gauss.c / gauss.h

Here the iterative Gaussian elimination algorithm `gaussian_-_elimination_iter` is implemented. The function takes as main arguments the coefficient matrix A, the recovered payload x, the encoded payload b and the index of the row `index_new` where the new packet was stored, e.g. if we receive a packet where many leading coefficients equal zero, we can already put this packet in its right position in the upper triangular matrix, instead of always inserting it in the first row. Thus we start Gaussian elimination at the diagonal element of row `index_new`, since we know that all rows above are already in a well shaped upper triangular form. While iterating over rows in the same column we also check if the value is zero and would skip that row then, since it is already well shaped. Gaussian elimination processes a sub square-matrix of the original matrix in each iteration. In this sub-matrix we put the row with the biggest first element in the first row and will perform the elementary row operations on the sub-matrix. For the case that this biggest first element is still zero, we will delete the whole row, since there is a high chance that this is an linear dependent vector (and anyway the inverse of the null-element is not defined). Otherwise Gaussian elimination wouldn't detect this linear dependency and the rank of A would be calculated wrong. However this situation barely occurs. The function returns the rank of A, which can be implemented as a simple counter of how many rows in A are non-zero, since A is always in upper triangular form. As soon as A has full rank and the node is configured as a sink, the node starts back substitution to recover x.

rnc.c / rnc.h

`rnc.c` implements all the functions to perform RNC based broadcasts. The function `init_rnc()` will initialize the nodes by the allocation of memory and setting the allocated memory to zero. Furthermore, each node will look up its mode (source, sink, relay, relay+sink) by its node-id and will open a broadcast connection. For the broadcast of messages we will use the broadcast implementation from the Contiki Rime stack, defined in `core/net/netstack.h`. We will set up an identified best-effort broadcast connection with `broadcast_open(...)` on a specific channel. We allocate memory for the struct `broadcast_conn` by declaring it as a static variable. The function `broadcast_recv(...)` is automatically called when a packet is received by setting it as the broadcast's designated callback function. Here we will parse the message type (RNC packet, NACK, NACK-reply) in order to determine how to process the incoming packet. In `rnc.h` the structure of a RNC-packet is defined as *struct rnc_pkt*. In order to send this packet over the broadcast channel we first copy it to the packetbuffer which is used by Rime's buffer management. This is done by `packetbuf_copyfrom(...)`. After that we can broadcast the packet by calling the `broadcast_send(...)` function. Various callback timers for the implementation of the timeouts will be used. In `start_rnc()` the source will initiate the RNC broadcast by sending out the first batch. Dependent on which value was set for `NUMBER_OF_BATCHES` in `params.h` the source will repeat sending batches in a `INTERVAL_BATCH` seconds interval. In turn, each packet in a batch is sent out after `INTERVAL_PKT` seconds. The packets sent by the source are uncoded which means that all coefficients are 0x00 except one which is 0x01. For debugging and demonstration purposes the payload of the packets is simply incremented for each byte in the respective GF.

Function `process_data()` will be called for all non-source nodes whenever we receive a RNC-packet or a NACK-reply and the rank of the coefficient matrix is still lower than K, i.e. the original payload from the source is not recovered yet. First, `process_data()` saves the incoming packet in the row which is determined by the leading zeroes of the coefficient vector. This row index will be returned and used in the Gaussian elimination algorithm to indicate on which submatrix to operate on. The old rank of the matrix will be saved and iterative Gaussian elimination will be executed and subsequently the old rank and the new rank will be compared. If the rank changes it is an indicator that new information was received and thus a relay can broadcast a new RNC-packet after a certain timeout. If a node's `timer_nack` times out, it will broadcast a NACK packet. This packet will contain the current batch-id and a byte, of which the bits are set if the respective packet is still missing. We hereby assume that $K \leqslant 8$ in order to fit into a byte. Nodes will also keep track of which encoded packets already arrived. We use a byte for that as well. So in order to determine if a node can respond to a NACK, it must only AND both bytes and check if the result is non-zero.

If a node fully recovered one batch it will only share its data, if asked by a NACK. For testing purposes, when a RNC-packet with a higher batch-id arrives, a node will discard the previous batch and clear all its memory. If the previous batch was not recovered completely until this point, it will print an error message and continues to process the next batch. In conclusion, in the current implementation we assume that batches are sent out after a big enough time interval in order to give all participating sinks the chance to recover the current batch.

util.c / util.h

Various other useful functions, e.g. minimum, maximum are defined here.

2.3.2 *Flooding*

In order to compare the RNC scheme with a basic flooding scheme we basically replace `rnc.c` and `rnc.h` with new files `flooding.c` and `flooding.h`:

```
~/contiki/core/net/seemoo-lab/flooding/
 |__ flooding.c
 |__ flooding.h
 |__ ...
```

The basic flooding scheme shares to a great extent the same functionalities as the RNC scheme. A source broadcasts several batches, each containing K packets. However all packets are unencoded and are sent one by one. If a relay receives a packet, it will store it and forward it with the same timeout and NACK mechanisms as in the RNC implementation. This should provide a mostly fair comparison between both schemes.

2.3.3 *Testbenches*

In ~/contiki/examples/sky/ we provide testbenches for both the RNC- and the flooding scheme. Both schemes will be simulated on a Tmote Sky. Tmote Sky is a wireless sensor module that provides high data rate and ultra low power and is a widely proven platform for wireless sensor systems deployments. Both files `test-rnc.c` and `test-flooding.c` provide the process which is required to generate the firmware for the Tmote Sky. Every process in Contiki should start with the PROCESS macro, which takes as arguments the name of the process structure and the name of the process. It is followed by an AUTOSTART_PROCESS macro which automatically starts the process given in the arguments when the module boots. After that we call the PROCESS_THREAD function which is used to define the protothread of a process. In this function we call the PROCESS_BEGIN macro which defines the beginning of a process. For the RNC implementation we can now initialize our RNC scheme by calling `init_rnc()`. Then we will wait one second to make sure every node is initialized before calling `start_rnc()` in order to start our broadcast. If powertrace output is enabled, we will additionally call `powertrace_start( CLOCK_SECOND * 2)` which prints the powertrace every two seconds on the console.[1, 15]

We use Cooja to simulate a small and a bigger network. Both RNC and flooding use the exact same setup. Cooja simulation files for those setups are located in ~/simulations/. Opening one of those files will automatically build the firmware with use of `test-rnc.c` or `test-flooding.c`. See chapter 4 for more information on the setup procedure.

2.4 PERFORMANCE EVALUATION

2.4.1 *Arithmetic Operations*

The precomputed look-up tables are essential in order to provide an energy efficient and fast implementation. The `gf_mul` and also the `gf_inv` functions are used extensively throughout the scheme. In order to determine the speedup between the optimized implementation with table lookups and the regular implementation we will determine

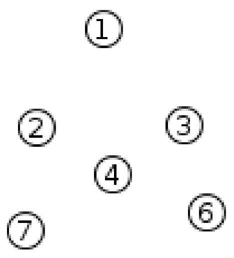


Figure 3: small network

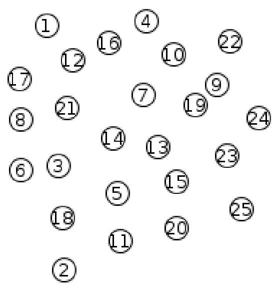
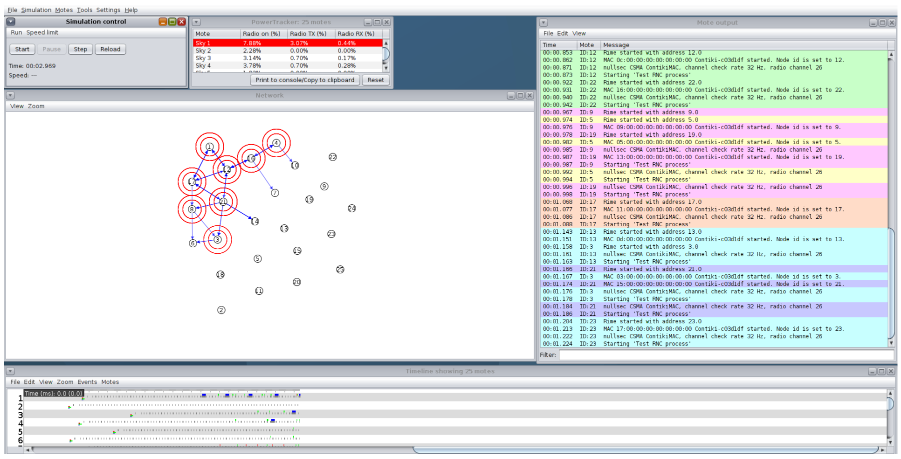


Figure 4: bigger network


File Simulation Motes Tools Settings Help
Simulation control
Run Speed limit
Start
Pause
Step
Reload
PowerTracker: 25 motes
Print to console/Copy to clipboard
Reset
Network
View Zoom
Mote output
File Edit View
Timeline showing 25 motes
File Edit View Zoom Events Motes

Time: 00:02.969
Speed: ---

Mote	Radio on (%)	Radio TX (%)	Radio RX (%)
Sky 1	7.88%	3.07%	0.44%
Sky 2	2.28%	0.00%	0.00%
Sky 3	3.14%	0.70%	0.17%
Sky 4	3.78%	0.70%	0.28%

```
Time       Mote   Message
00:00.853  ID:12  Rime started with address 12.0
00:00.862  ID:12  MAC 0c:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 12.
00:00.871  ID:12  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:00.873  ID:12  Starting 'Test RNC process'
00:00.922  ID:22  Rime started with address 22.0
00:00.931  ID:22  MAC 16:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 22.
00:00.940  ID:22  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:00.942  ID:22  Starting 'Test RNC process'
00:00.967  ID:9   Rime started with address 9.0
00:00.974  ID:5   Rime started with address 5.0
00:00.976  ID:9   MAC 09:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 9.
00:00.978  ID:19  Rime started with address 19.0
00:00.982  ID:5   MAC 05:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 5.
00:00.985  ID:9   nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:00.987  ID:19  MAC 13:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 19.
00:00.987  ID:9   Starting 'Test RNC process'
00:00.992  ID:5   nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:00.994  ID:5   Starting 'Test RNC process'
00:00.996  ID:19  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:00.998  ID:19  Starting 'Test RNC process'
00:01.068  ID:17  Rime started with address 17.0
00:01.077  ID:17  MAC 11:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 17.
00:01.086  ID:17  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:01.088  ID:17  Starting 'Test RNC process'
00:01.143  ID:13  Rime started with address 13.0
00:01.151  ID:13  MAC 0d:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 13.
00:01.158  ID:3   Rime started with address 3.0
00:01.161  ID:13  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:01.163  ID:13  Starting 'Test RNC process'
00:01.166  ID:21  Rime started with address 21.0
00:01.167  ID:3   MAC 03:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 3.
00:01.174  ID:21  MAC 15:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 21.
00:01.176  ID:3   nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:01.178  ID:3   Starting 'Test RNC process'
00:01.184  ID:21  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:01.186  ID:21  Starting 'Test RNC process'
00:01.204  ID:23  Rime started with address 23.0
00:01.213  ID:23  MAC 17:00:00:00:00:00:00:00 Contiki-c03d1df started. Node id is set to 23.
00:01.222  ID:23  nullsec CSMA ContikiMAC, channel check rate 32 Hz, radio channel 26
00:01.224  ID:23  Starting 'Test RNC process'
```

Filter:

Figure 5: Cooja simulator. Cooja provides different tools to debug or test the implementation. With the simulation control we can start, pause or reload a simulation. With the PowerTracker we can access the Radio Duty Cycle of an individual node. In sensor network window, nodes can be created, rearranged or deleted. Wireless traffic can be visualized in the network window and in the timeline. Furthermore nodes can print status messages to the console

the cycles of each implementation on a regular machine rather than in the simulator. The clock resolution of the simulator is not sufficient for measurements. Table 1 shows the different CPU cycle measurements on a middle class desktop CPU. We can observe that for GF(256) and for GF(16) the table look-ups are way faster and thus more efficient. Especially the regular calculation of the multiplicative inverse consumes a lot of time compared to the look-ups. We would assume a similar speedup on a sensor node hardware architecture. Furthermore, observing the messages sent and the not measurable computation times, we conclude that the computation times are negligible in comparison to the transmission times.

Operation	GF(256)	GF(16)
`gf_mul_regular`	121	54
`gf_inv_regular`	1262	126
`gf_mul_table`	30	30
`gf_inv_table`	29	29

Table 1: Cycles counts of `gf_mul` and `gf_inv`.

2.4.2 *RNC vs. Flooding*

We will compare both schemes regarding energy consumption and latency, i.e. the time between when the source starts sending out a batch and the full recovery of a batch at a sink which is located at the edge of the network. With these result we can also derive the reliability of both schemes. For both the small network (see Fig. 3) and the bigger network (see Fig. 4) we will consider a payload size $M = 100$ bytes, as well as $K = 6, 7, 8$ and calculations in GF(256), GF(16) and GF(2) for RNC. Table 3 compares the average latency of both schemes. To provide a fair comparison we also execute the flooding scheme for all different GF and average the results. However the latency of flooding should be independent of the chosen GF. For the small network we chose node 6 and 7 to be sinks, whereas for the bigger network we chose the outer nodes 2 and 24 as sinks. Source node is always node one. Furthermore we choose the following fixed timeout delays:

- delay per batch: K sec
- delay per packet: 0.25 sec
- delay RNC-packet: 0.125 sec - 0.25 sec
- delay NACK: 0.5 sec - 1 sec
- delay NACK-reply: 0.125 sec - 0.25 sec

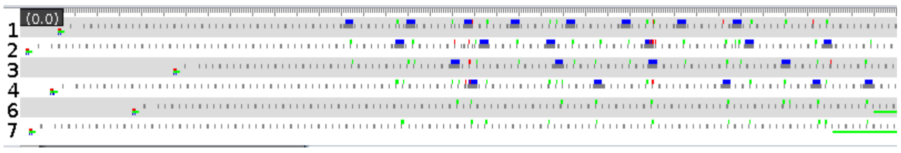


Figure 6: messages sent in RNC scheme with $K = 8$.

For the energy consumption we will consider the average Radio Duty Cycle (RDC) of the network. We observe three values in %: Radio on / Radio TX / Radio RX, measured with the PowerTracker tool

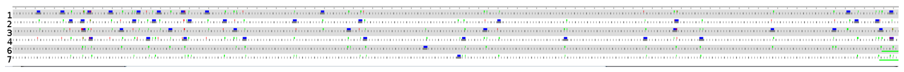


Figure 7: messages sent in flooding scheme with $K = 8$.

		small network		bigger network	
K	GF	RNC	flooding	RNC	flooding
6	256	1642	5385	9495	6255
7	256	1782	7812	5145	9636
8	256	2290	10453	4672	11270
6	16	1635	5385	9525	6255
7	16	3800	7812	9527	9636
8	16	2260	10453	5700	11270
6	2	3999	5385	21550	6255
7	2	5614	7812	21882	9636
8	2	6191	10453	20642	11270

Table 2: comparison of the average latencies for the small and bigger network in milliseconds

at the point when the first sink recovers a full batch. We can use a LED breakpoint to stop at exactly this point in time, i.e. when the LED is turned on to indicate a successful recovery. Since these points in time differ for the flooding- and the RNC scheme we will normalize the RDC values with help of Table 3 to those of RNC. Thus, the RDC for flooding (denoted by *) is calculated as if the latency were equal to RNC.

As we can see in both tables, in the smaller network scenario RNC outperforms flooding in each parameter set. The latency and the radio duty cycles are lower for every K and GF. We can also observe that the latency increases the smaller we choose our GF. However, for GF(2) the latency is really high, for both the small and the bigger network. This is due to too many linear dependent vectors. Nodes can only chose elements in $\{0, 1\}$, so with a probability of 50% we choose zero and do not encode the payload. Therefore a lot of NACKs have to be sent. In the bigger network we observe that for $K = 7, 8$ and GF(256) and GF(16) RNC still outperforms flooding, while for $K = 6$ flooding has a lower latency and energy consumption. This might indicate some bad settings for the timeout values. Additionally, both schemes provide a 100% reliable data transmission, if the batch delay is chosen accordingly.

		small network		bigger network	
K	GF	RNC	flooding*	RNC	flooding*
6	256	6.9/2.1/1.1	21.9/6.7/2.9	6.6/1.6/1.0	4.6/1.2/0.7
7	256	7.5/2.2/1.2	28.3/7.7/4.0	8.4/2.1/1.6	14.2/3.9/2.3
8	256	8.3/2.8/1.3	26.8/6.8/3.1	8.3/2.4/1.5	18.4/5.1/3.0
6	16	6.9/2.1/1.1	21.4/6.1/2.8	6.8/1.7/1.1	4.7/1.2/0.76
7	16	6.8/1.9/1.0	12.2/3.1/1.4	7.0/1.8/1.1	7.3/1.9/1.2
8	16	7.6/2.4/1.2	27.1/6.8/3.2	7.4/2.0/1.2	15.2/4.2/2.4
6	2	6.9/2.0/1.1	7.9/1.7/0.9	6.3/1.4/0.9	2.1/0.5/0.3
7	2	6.9/1.9/1.0	8.8/2.3/1.2	6.5/1.5/1.0	2.9/0.7/0.4
8	2	6.6/1.7/1.0	10.5/2.8/1.4	6.9/1.6/1.1	3.6/0.9/0.5

Table 3: comparison of the Radio Duty Cycles for the small and bigger network

3 CONCLUSION

In the last chapter we demonstrated a basic Random Network Coding broadcast scheme with an NACK mechanism, that ensures 100% reliability. We have shown that the overhead produced by the linear combinations and the Gaussian elimination is negligible due to efficient field computations and an optimized Gaussian elimination algorithm. Furthermore, we demonstrated that RNC can outperform flooding if we chose a high enough K in order to ensure reliability and a big enough Galois Field to ensure a high chance of linear independent vectors.

3.1 FUTURE WORK

This lab set the basis for further development of efficient Random Network Coding. There are still unimplemented features which can be added to improve the efficiency. Two or more GF elements from a field smaller than GF(256) could be encoded into one byte to increase the data rate. Furthermore the currently implemented NACK mechanism needs a fine adjustment of the fixed timeout delays, especially for different values of K. A goal is to make this more independent. The scheme should then also be tested in a less ideal setting, e.g. a higher interference range, and might be compared against more sophisticated approaches besides naive flooding.

4 USER DOCUMENTATION

REQUIREMENTS

- InstantContiki 3.0 [9]
- Oracle VM VirtualBox or VMware Player

INSTALLATION

1. start InstantContiki 3.0
2. overwrite ~`/contiki/` with `RNC_JanSturm/contiki/`
3. start Cooja:
   - cd ~`/contiki/tools/cooja/`
   - ant run (if error message: git submodule update –init
4. open a simulation
   - File -> Open simulation -> Browse
   - Choose one of the simulations from `RNC_JanSturm/simulations`
   - Click on "Start" in the Simulation Control window

The parameters for the respective scheme can be adjusted in

- ~`/contiki/net/seemoo-lab/rnc/params.h`
- ~`/contiki/net/seemoo-lab/flooding/params.h`

BIBLIOGRAPHY

[1] *Contiki broadcast example*. URL: `http://anrg.usc.edu/contiki/index.php/Broadcast_Example`.

[2] *Contiki*. URL: `http://www.contiki-os.org/`.

[3] *Finite Fields*. URL: `https://www.rocq.inria.fr/secret/Anne.Canteaut/MPRI/ff.pdf`.

[4] C. Fragouli, J. Widmer, and J. Y. Le Boudec. "A Network Coding Approach to Energy Efficient Broadcasting: From Theory to Practice." In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. 2006, pp. 1–11. DOI: `10.1109/INFOCOM.2006.45`.

[5] *GF table*. URL: `https://crypto.stackexchange.com/questions/12956/multiplicative-inverse-in-operatornamegf28`.

[6] *Guassian Elimination*. URL: `http://fourier.eng.hmc.edu/e176/lectures/NM/node5.html`.

[7] I. H. Hou, Y. E. Tsai, T. F. Abdelzaher, and I. Gupta. "AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks." In: *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. 2008. DOI: `10.1109/INFOCOM.2008.211`.

[8] Jonathan W. Hui and David Culler. "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale." In: *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*. SenSys '04. Baltimore, MD, USA: ACM, 2004, pp. 81–94. ISBN: 1-58113-879-2. DOI: `10.1145/1031495.1031506`. URL: `http://doi.acm.org/10.1145/1031495.1031506`.

[9] *Instant Contiki 3.0*. URL: `https://sourceforge.net/projects/contiki/files/Instant%20Contiki/Instant%20Contiki%203.0/InstantContiki3.0.zip/download`.

[10] *Network Coding: An Introduction*. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.422.5525&rep=rep1&type=pdf`.

[11] *Russian Peasant Algorithm*. URL: `http://lafstern.org/matt/col3.pdf`.

[12] *Russian Peasant C-Code*. URL: `https://en.wikipedia.org/wiki/Finite_field_arithmetic#C_programming_example`.

[13] *Smartdust* . URL: `http://www-bsac.eecs.berkeley.edu/smartdust/`.

[14] *The Laws of Cryptography: The Finite Field GF(28).* URL: `http://www.cs.utsa.edu/~wagner/laws/FFM.html`.

[15] *Tmote Sky.* URL: `https://wirelesssensornetworks.weebly.com/blog/tmote-sky`.