Data block fragmentation with packet loss tolerance. Possible application to partial firmware upgrade over the air

Authors:
N. Sornin (Semtech)

Version:Draft 0.3
Date: 2015 Dec

Contents

Tables

No table of figures entries found.

Figures

1 Revisions

Author	Date / Rev	
Nsornin	1/26/2015 V0.0	First draft
Nsornin	4/24/15 v0.1	Minor typo corrections, upload to portal
Nsornin	12 nov 2015 v0.2	Corrected headers & footers

2 Introduction

This document is a proposal describing ways to:

- Split a long data block into smaller fragments
- Rearrange those fragments to add error correction capabilities
- Transmit those fragments to the receiver(s) . This transmission may include multicast or unicast downlinks or uplinks , with an unpredictable loss rate
- Recover and reassemble the original data block from the received fragments even if some of those have been lost
- Acknowledge the reception of the original data block
- Authenticate the data block
- Request status and missing blocks to the receiving end.

This document also proposes a process for partial firmware upgrade over the air of a large quantity of end-device's through a multicast block distribution.

Fragmentation may be used for many different applications, for example:

- Broadcasting a firmware upgrade to a group of end-devices (Network -> end-devices downlink multicast)
- Fragmenting a huge data block in several smaller messages before sending it up to the network (end-device -> Network uplink) and make sure all the data has been successfully received. Example: A sensor collects frequent data for a long time, and then compresses it into one huge block that is sent using fragmentation, as soon as the server is able to reconstruct the full block the device receives a notification and stops transmitting.
- Creating redundancy to improve the delivery probability of an uplink. Using this method is much more efficient than simply repeating the message several time.

This fragmentation method sits in the Application layer on top of the LoRaMAC layer (class A, or A+B or A+C) and only use applicative payload. It only requires means to send and receive payloads. This method uses a reserved range of application port. Any data received on a port belonging to this range will be processed as fragmented data. Similarly, any data sent on a port belonging to this range will be processed as fragmented data on the other end of the link. The reserved port range is end-device specific and must be communicated to the network server out-of-band during device activation.

The coding scheme used is directly derived from the 1963 thesis of Robert Gallager describing Parity-Check code: This thesis can be accessed at
http://www.inference.phy.cam.ac.uk/mackay/gallager/papers/ldpc.pdf

3 Payload format

The payload exchanged between the frag/defrag application and the MAC layer may contain either a data fragment or a fragmentation command depending on the value of the first byte called FragCtrl

If FragCtrl <128 then the payload content is:

Size (bytes)	1	1	0:MaxAppPl-2
Payload	FragCtrl	FragDesc	P_M^N

Where the FragCtrl byte contains:

Bit#	7	[6:0]
FragCtrl	Always 0	N

And the FragDesc byte contains:

Bit#	[7:6]	[5:3]	[2:0]
FragDesc	fragID	FragNb	FragPerFrame

fragID ID is the identifier of the current fragmentation session. It is incremented (modulo 4) each time a new fragmentation session is started. This allows the receiver to discriminate between two fragmentation sessions using the same parameters in case many consecutive frames have been lost. We do not want the receiver to try to reassemble together some fragments belonging to two different fragmentation session

FragNb encodes in a compressed form the number of fragments (of the current fragmentation session) into which the data block was divided. The actual number of fragment used in this fragmentation session is M given by the following table:

FragNb	M
0	32
1	40
2	48
3	56
4	64
5:7	rfu

M is the baseline number of fragments composing the data block to be transported. More than M fragments may actually be transmitted to add redundancy and packet loss robustness. M is also not directly related to the number of frames that will be physically sent over the air, as each frame may transport several fragments as explained later in this document.

N is the index of the first coded fragment transported in the frame.

The encoded data transported in the fragment N is noted P_M^N . Note that when adding redundancy the transmitter will transmit more than M fragments so N may be greater than M

FragPerFrame encodes in a compressed format the number of fragment transported in each frame payload (P_M^N). For algorithmic performance reasons that will be explained later in the document, the minimum number of fragment has been set to 32. However transmitter may want to split the data block across fewer than 32 frames. So each frame can actually contain multiple small fragments.

The number of fragments per frame (FPF) is given by the following table:

FragPerFrame	FPF
0	1
1	2
2	4
3	8
4	16
5:7	RFU

Example:

Field	FragCtrl	FragDesc
Value	0x20	0x4A

The FragCtrl byte contains:

Bit#	7	[6:0]
FragCtrl	0 means this is a coded fragment	N
	0	0x20

And the FragDesc byte contains:

Bit#	[7:6]	[5:3]	[2:0]
FragDec	fragID	FragNb	FragPerFrame
	1	1	2

This means that this is the fragment 32 (0x20) out of 40, of fragmentation session 1. Each frame transports 4 fragments. This means that the payload of this frame contains fragments 32,33,34 and 35.

If FragCtrl>=128 then the payload contains a fragmentation command:

Size (bytes)	1	0..3
Payload	FragCtrl	FragParam

Where the FragCtrl byte contains:

Bit#	7	[6:0]
FragCtrl	Always 1	FragCommand

The FragCommand field designates the command transported. Some commands might require additional parameters in that case those parameters are in the FragParam field.

4 Fragmentation control messages

Frag Command	Transmitted by		**Description**
	Block sender	Block receiver	
0x01		X	**Data block reception acknowledge** Used by the receiver to signal the transmitter than the full block was successfully received and reconstructed
0x02		X	**Data block authentication request** Used by the receiver to signal the transmitter that the block was successfully received and request authentication
0x03	X		**Data block authentication answer** Used by the transmitter to confirm the data block authenticity
0x04	X		**Fragmentation session status req** Request all receivers of the fragmentation session to report their current defragmentation status.
0x05		X	**Fragmentation session status ans** Sends back the status
0x05 – 0x7F			RFU

4.1 Data block reception acknowledge

Size (bytes)	1
FragParam Payload	fragID

This command may be sent by an end-device or a server.
The “data block reception acknowledge” command is sent by the fragmentation receiver and signals to the transmitter that the full block corresponding to the fragmentation session fragID was reconstructed successfully.

If the data block is broadcasted to many end-devices using multicast, then the delay between the reception of the last fragment that enables reconstruction of the full block and the transmission of the “data block reception ack” frame must be randomized to avoid uplink massive collisions. The randomization interval is a function of the maximum number of end-devices belonging to the multicast group targeted in this fragmentation session. The more end-devices, the longer the interval. This interval can either be an application specific parameter (the size of the multi-cast group is known before-hand) or can be provided in the data block itself, see “recommended data block format” chapter.

4.2 Data block authentication request

Size (bytes)	1	4	0:..
FragParam Payload	fragID	MIC	Custom payload

This command can only be sent by an end-device to an application server.
The “data block authentication request” command is sent by the fragmentation receiver (the end-device) and signals to the transmitter (the server) that the full block corresponding to the fragmentation session fragID was reconstructed successfully. It also asks the transmitter to confirm the authenticity of this data block by computing a cryptographic 32 bits hash of the data block using the device’s AES128 application session key “AppKey”. The MIC field is computed as follow :

cmac = aes128_cmac(**AppSKey**, B_0 | *msg*)
MIC = *cmac*[0..3]

whereby the block B_0 is defined as follows:

Size (bytes)	1	9	4	2
B0	0x49	13 x 0x00	Unicast DevAddr	len(*msg*) in bytes

And msg = [B1 | B2 |… |Bm] , the concatenation of all the uncoded fragments.

> Note : The MIC must be computed using the **UNICAST** Application Session Key of the end-device. **Do not use** a multicast Application Session Key as this key may be compromised in one device of the multicast group has been compromised.

The optional “Custom payload” field can be used to transport any additional application layer information related to the fragmentation session. For example , if fragmentation is used to distribute a firmware upgrade then this field can be used to transport a MIC or a CRC of the memory area which is going to be upgraded to ensure that the memory content being replaced is exactly as it should be and avoid to upgrade out-of-date end-devices.

4.3 Data block authentication answer

Size (bytes)	1
FragParam Payload	FragAuthAns

Where:

bits	7:3	2	1:0
FragAuthAns field	RFU	1=authentication successful	FragId

		0=failed	

The data block authentication answer is sent by the transmitter of the fragmentation session upon reception of the “Data block authentication request” command.
The transmitter computes the same hash on the data block using the receiver’s application session key and compares it to the signature in the “Data block authentication request” message. If the two signatures match then the data block is authentic and hasn’t been altered during the fragmentation and transport steps.

4.4 Fragmentation session status Req

Size (bytes)	1
FragParam Payload	FragStatusReqParam

Where:

bits	7	6	5:2	1:0
FragStatusReqParam field	RFU	Participants	Answer delay spread	FragId

Used by the fragmentation transmitter to request receiver devices to report their current defragmentation status.

The receivers (in the case of a multi-cast) should not answer this request all at the same time because this would potentially generate a lot of collisions. The receivers must therefore spread randomly their response of a given time period.
The answer delay spread parameter encodes the length of this response interval. The response interval should be $2^{Answer\_delay\_spread}$ seconds.
The valid range goes from 1sec (immediate answer allowed, typically used for unicast fragmentation) with answer_delay_spread = 0 to 4096 seconds with answer_delay_spread = 12.

The “participants” bit signals if all the fragmentation receivers should answer or only the ones still missing fragments.

Participant bit value	0	1
	Only the receivers still missing fragments must answer the request	All receivers must answer , even those who already successfully reconstructed the data block

4.5 Fragmentation session status Ans

Size (bytes)	1
FragParam Payload	FragStatusAnsParam

Where:

bits	7:2	1:0
FragStatusAnsParam field	MissingFrag	FragId

Used by the fragmentation receiver to report its defragmentation status for the fragmentation session FragId.

MissingFrag is the number of independent coded fragments still required before being able to reconstruct the data block. In the case where the block was already successfully reassembled this field should be 0.
As described in the “Fragmentation session Status Req” command, the receivers must respond with a pseudo-random delay as set by the answer_delay_spread parameter transmitted in the request. This pseudo-random delay can be derived from a pseudo-random number generator seeded with the device’s devAddr (unicast network address of the device). This guarantees that all devices will follow different pseudo-random sequences.

5 Partial firmware upgrade over the air

Partial firmware upgrade over the air is one of the use cases which may use fragmentation.

5.1 Pre-requisite for FUOTA

Only **partial** firmware upgrade is considered to be doable using the LoRaWAN protocol. The block sizes that can be reasonably distributed are between 200 to 2kBytes:
This means that only a few portions of the program FLASH memory may be upgraded, not the entire FLASH. Therefore the code must be linked specifically to make that possible. The following recommendations should be considered:

- Manually specify to the linker the base address for every function that you will want to patch. If possible align those with pages of the FLASH memory
- Make sure that there is some empty memory space between each functions, so that a function can slightly grow without having to move the entire memory content
- Make the main function loop as short & simple as possible and essentially a list of function calls. This way the main.c code which describes the essential state machine of the end-device will be easily upgradable and its behavior will be easy to change without transporting too much code.
- Your application must handle the defragmentation and be able to copy the received blocks into the main program FLASH memory, then reset the device.
- Never try to modify the portion of the FLASH memory that contains the “memory upgrade code”

The upgrades Server must keep track of what firmware patches are installed on which devices. Assume that a few % of the devices will never be able to perform the upgrade for various reasons. Therefore one cannot consider that all devices are running the same firmware version. Your application server must be able to cope simultaneously with all versions of the device firmware that where distributed.

5.2 FUOTA process

The software patch block is assembled using the recommended format described in 5.3 by the upgrade server. This block is fragmented and distributed using a multicast session to all targeted end-devices (or through normal unicast traffic with each end-devices).
When an end-device has received enough fragments and reconstructed the complete patch block the following steps happen:

1. The end-device verifies the **CRC** of the block and compares it to the ***CRC*** field. If the CRC is wrong the end-device may optionally try to replace one or several fragments by additionally received coded fragments and invert again until a correct CRC is found. This process is not described in this document. The probability of a corrupted block is extremely low has each frame transported by the LoRaWAN protocol has a 32bits MIC field.
2. The end-device computes a 32 bit CRC over the intervals of its program FLASH memory that are going to be replaced: CRC([StartAddr1:StartAddr1+BlockLen1] | [StartAddr2:StartAddr2+BlockLen2] | …)
3. The end-device computes a random delay in [1: be $2^{Answer\_delay\_spread}$] seconds
4. The end-device sends the “Data block authentication request” command to the fragmentation server. This command contains the computed memory CRC as an optional payload field as described in 4.2.

5. If no answer is received, the end-device must retransmit the “Data block authentication request” frame, waiting for a random delay in the in [1: be $2^{Answer\_delay\_spread}$] seconds interval between each retransmission.
6. The server checks the authenticity of the data block by computing the same MIC than the end-device and also checks that the reported memory CRC corresponds to the expected value. If both are correct the server sends back a “Data Block authentication answer” with flag set to 1. If something is wrong the server sends the answer with a flag set to 0. If the memory CRC is wrong this means that the end-device current firmware is not the one expected, therefore it would be dangerous to patch it as this would put the device in a probably unknown state.
7. If the authorization flag is set, the end-device stores the *PatchNb* field in the list of installed patches and starts the firmware patch copy process.
8. The end-device reboots.
9. When the device has optionally rejoined the network (for OTA devices), the device’s application layer may start by sending a message containing the list of patches installed. This way the application server knows the exact firmware version running inside the device. Alternatively, the end-device’s application may be designed to react to a specific application downlink command requesting the firmware version, and answer with the list of patches installed.

5.3 Recommended data block format for FUOTA

This chapter provides a recommend format for **large** (200bytes to 3kbytes) data block broadcasted to many end-devices. The typical use case targeted is partial firmware upgrade over the air. The format can be adapted for application specific purposes. This is only intended as a guideline for the software designer.

Field name	Length (bytes)	description
AnswerDelaySpread	1	The answer delay spread parameter encodes the length of the response interval. The response interval should be $2^{Answer\_delay\_spread}$ seconds. The valid range goes from 1sec (immediate answer allowed, typically used for unicast fragmentation) with answer_delay_spread = 0 to 4096 seconds with answer_delay_spread = 12.
PatchNb	1	Identifier of the current software patch
StartAddr1	2	Start address in the memory of the first patch block

BlockLen1	2	Length in byte of the first patch block
Block1	BlockLen1	Binary content of the first patch block to be copied into the FLASH memory of the device , starting at StartAddr1
StartAddr2	2	Start address in the memory of the second patch block
BlockLen2	2	Length in byte of the second patch block
Block2	BlockLen2	Binary content of the second patch block to be copied into the FLASH memory of the device , starting at StartAddr1
CRC	4	32bits CRC of the full block once assembled. Used by the end-device to check that the reconstructed block is error free before requesting the authorization to install.
ZeroPadding	N	If required , in order for the full block length to be a possible combination of fragment number x fragment size

6 Fragment error coding

The initial data block that needs to be transported must first be fragmented into M data fragments of arbitrary but equal length. The length of those fragments has to be chosen to be compatible with the maximum applicative payload size available.
The actual applicative payload length will be:

fragLen x FPF + 2 bytes

Where fragLen is the length of each fragment, FPF is the number of fragments per frame, plus 2 bytes of fragmentation header (containing FragCtrl and FragDesc)

Those original data fragments are named uncoded fragments and are noted Bn

The full data block to be transported consists therefore of the concatenation of the Bn uncoded fragments [B1 : B2 :… :Bm]

The coded fragments are noted $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ and are derived from the uncoded fragments.
$\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ is the Nth coded fragment from a fragmentation session containing M (B1 to Bm) uncoded fragments.
To allow the original uncoded fragments reconstruction on the receiving end of the link even in presence of arbitrary packet loss, the fragmentation application adds redundancy. Therefore N might be greater than M, meaning that the sender may send more coded fragments than the total number of uncoded fragments to enable the reconstruction on the receiving end in presence of packet loss. The ratio between M and the number of actually sent coded fragments is called coding ratio and noted CR

The coded fragments $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ are constructed by performing a bit per bit Xor operation between different subset of the uncoded fragments. The Xor operator is noted +.

Each coded fragment is defined as :

$$\mathrm{P}_{\mathrm{M}}^{\mathrm{N}} = C_N^1.B_1 + C_N^2.B_2 + .. + C_N^M.B_M$$

Where C_N^i is a function of M,N and i and is either 0 or 1. The M parameter is not systematically added to the notation for clarity.

- $0.B_1$ is a word of the same length than the fragment B_1 with all bits = 0.
- $1.B_1$ is equal to B_1

The binary vector $C(N,M) = [C_N^1, C_N^2, \ldots, C_N^M]$ of M bits is a function of M and N and is given by a function matrix_line(M,N) which will be described later.

It is sufficient to know that this function generates a parity check vector containing statistically as many zeros as ones in a pseudo-random order.
The parity check matrix, as defined by Gallager in his 1963 thesis, is an MxN matrix containing the C_N^i on column i and line N.
The following picture illustrates an example of such a matrix generated by the proposed function for M=64 and with a coding ratio CR=1/2 , this matrix has 64/CR = 128 lines. Id , it allows the creation of 128 coded fragments from the 64 original uncoded fragments.

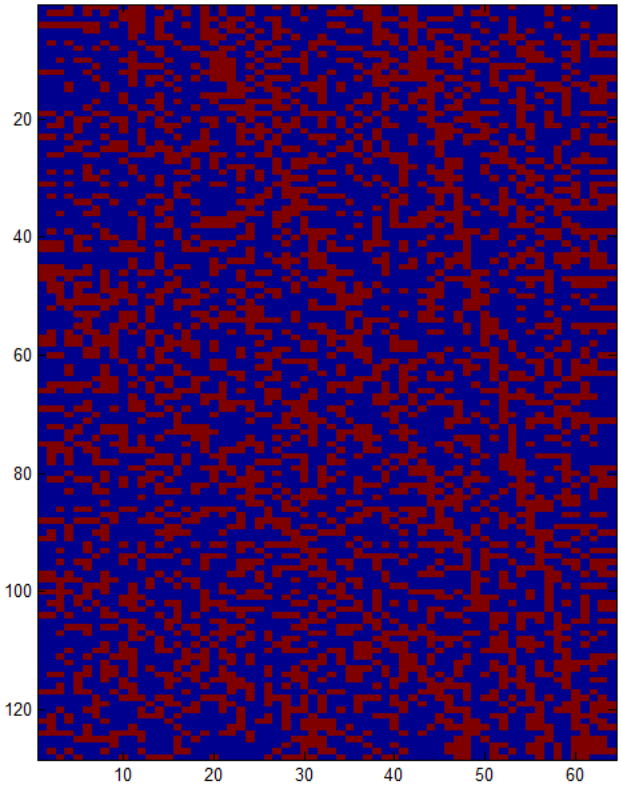


Figure 1 : 64x128 parity check matrix

The coded fragment P_M^N is therefore the bit-wise Xor of the uncoded fragment B_i such that C_N^i is non-zero. The coded fragments have exactly the same bit length than the uncoded fragments.

Step by Step encoding example:
The transmitter must send 256 bytes with a coding ratio of ½ (basically allowing 50% Packet Error Rate on the radio link) . For this purpose the 256 data block will be segmented in 32 fragments of 8 bytes , each frame will transport 4 fragments , therefore 256 / 8 / 4 / CR = 16 frames will be generated. We will see that the receiver will be able to decode as soon as it receives 8 or 9 frames out of the 16 (depending on the exact combination of frames lost).

First split the 256 bytes into 32 uncoded fragments of 8 bytes each , B1 to B32.

To generate the first coded fragment.
First generate the first line of the parity check matrix by calling C = matrix_line(1,32)
Then perform a bitwise Xor operation between all the uncoded fragments corresponding to a 1 in the C parity check vector.

In this case C(1,32) = 01011110111001001010100000100000 , there the first coded fragment P_{32}^1 = B2 + B4 + B5 + B6 + B7 + B9 + B10 + B11 + B14 + B17 + B19 + B21 + B27
Where + is the bit-wise Xor operator

Compute identically the coded fragments $P_{32}^{2}, P_{32}^{3}, \text{and } P_{32}^{4}$, then assemble the first frame by pre-pending the fragmentation header and concatenating the 4 first coded fragments.

Field	FragCtrl	FragDesc	P_{32}^{1}	P_{32}^{2}	P_{32}^{3}	P_{32}^{4}
Payload	0x01	0x02	8 bytes	8 bytes	8bytes	8bytes

In our example, assuming the data block to be transported contains the following 256 bytes [0,1,…,254,255] , then the first coded fragment will contain:
[120 121 122 123 124 125 126 127]

The FragCtrl and FragDesc field signal that this is the first coded fragment out of 32 from fragmentation session 0, using 4 fragments per frame.

Proceed identically for the next 15 frames. The 16th frame will look like:

Field	FragCtrl	FragDesc	P_{32}^{60}	P_{32}^{61}	P_{32}^{62}	P_{32}^{64}
Payload	0x10	0x02	8 bytes	8 bytes	8bytes	8bytes

7 Fragment decoding and reassembling

The receiver of a fragmentation session must perform the following operations.

For each frame received, break it into individual coded fragments, because a frame may carry multiple fragments. The number of fragments carried in the frame is indicated in the fragmentation header as well as the index of the first fragment.

The receiver also needs to create a null binary A = MxM matrix structure in his memory.

Then process those fragments one by one.

1. For each new fragment $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$, first fetch the corresponding line of the parity check matrix : C=matrix_line(N,M)
2. Proceed from left to right along the C vector ($i\ varying\ from\ 1\ to\ M$) : For each entry C_i equal to 1, check if the line i of the matrix A contains a 1 in row i. If yes, perform a Xor between line i of matrix A "A(i)" and the vector C and store the result in C. Also perform a xor between $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ and the coded fragment stored at position i in the fragment memory store S_i and update $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ with the result.
3. Once this process is finished there are two options:
    a. Either C now contains only zeros, in that case just get rid of the coded fragment $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$; it isn't bringing any new information
    b. The vector C is non-null : write it in the matrix A at the line i corresponding to the first non-zero element of C. Also add the modified $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ fragment to the memory store at position i : S_i
4. Loop to 1 until all lines of the matrix A have been updated. The matrix A will have only 1's on its diagonal and will be a triangular matrix with only 0's on the lower left half. The fragment memory store will contain exactly M fragments.
5. Starting from matrix line i = M-1 down to 1, fetch the i^{th} line of matrix A : A(i). The line A(i) has a 1 at position i and only zeros on the left. For any 1 at position j> i perform a xor between S_i and S_j and update S_i with the result.
6. The fragment memory store now contains the original uncoded fragments $S_i = B_i$
7. Reassemble the data block by concatenating all the uncoded fragments. If the fragment memory store is actually allocated as a continuous memory range, then this step is not even necessary, because the original data block consists of $S_1 : S_2 : \ldots : S_M$ where : represents the concatenation operator.

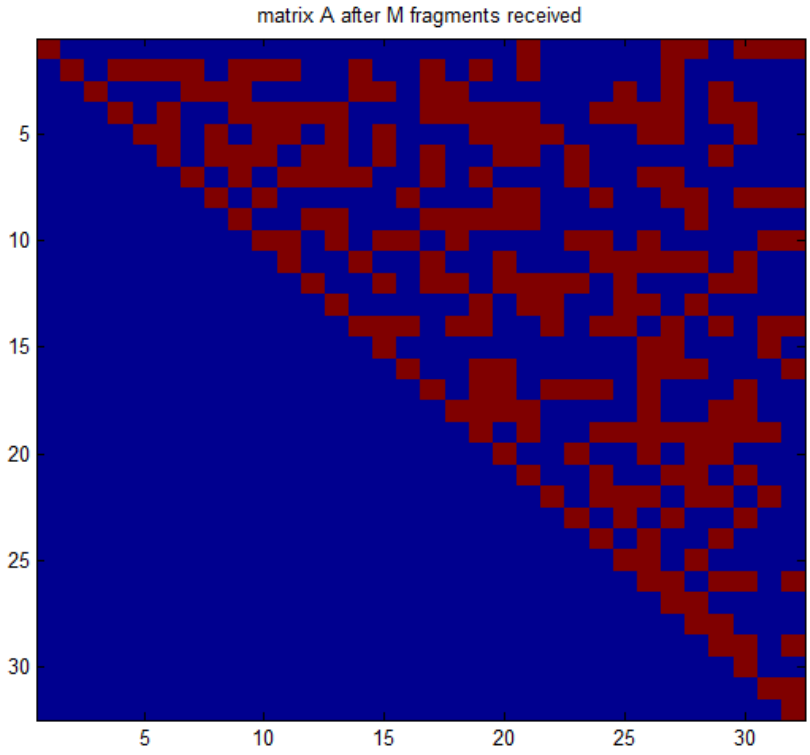
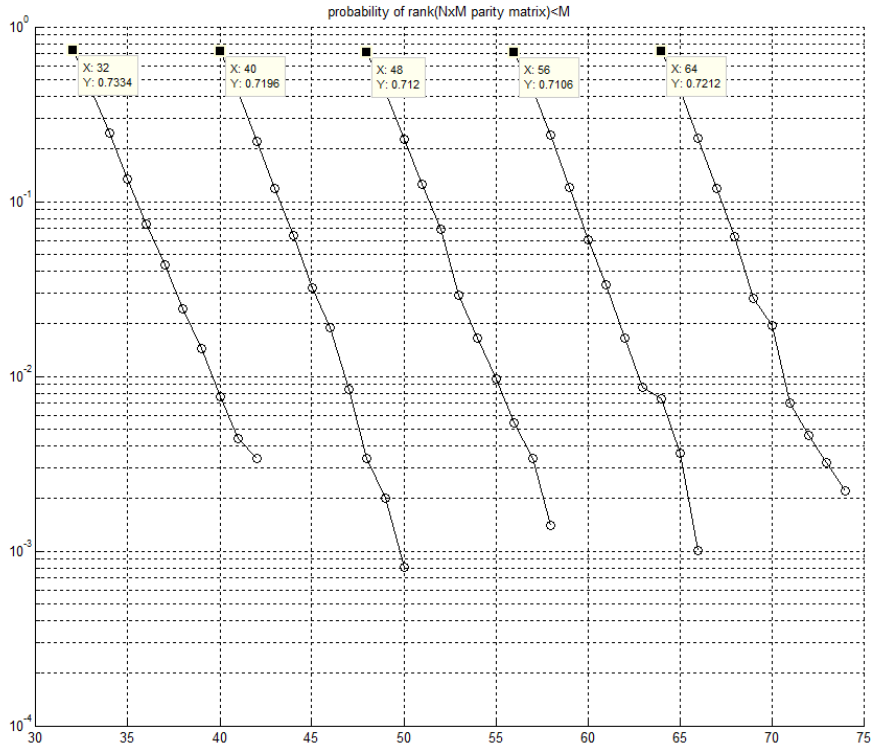


Figure 2 : 32x32 matrix A built during decoding process

8 Performance of the coding scheme.

As described in Gallager's thesis Parity Check codes have a non-zero statistical overhead independent of the coded word length. In our case the word length used is M. The actual overhead depends on the way the parity check matrix is built. To be able to reconstruct the uncoded fragments the receiver must receive at least M linearly independent coded fragments. Said in another way, the parity check matrix reconstructed by the receiver based on the fragments received must be of rank M.
This condition is fulfilled ideally as soon as M coded fragments have been received. But sometimes, those M received first fragments are not all independent and the matrix resulting rank is <M. in that case, more coded fragments need to be received until the rank of the parity check matrix becomes M.

The following graph shows the probability of the matrix being of rank <M with a number of received fragment varying from M to M+10. The 5 curves corresponds to M=32/40/48/56/64 which are the number of uncoded fragments used in this proposal.



It can be seen that when the number of coded fragment received equals M, the matrix cannot be inverted (is not of rank M) 70% of the cases. But this probability falls very rapidly with a few additional received fragments. With M+7 fragments the matrix can be inverted in 99% of the occurrences. In takes in average M+2 coded fragments to recover the original data block. The proposed fragmentation therefore works better (with a lower statistical overhead) with a larger number of fragment. For small data blocks this may lead to very short fragments, this is why the proposed protocol allows transporting several fragments per frame ,and the minimum number of fragments is set to 32.

9 End-device memory requirement

The following table gives the decoding RAM requirements for an optimized end-device implementation expressed in Bytes.

Nb of fragments	Fragment memory store	Parity matrix
32	32*fragLen = X	80
40	40* fragLen = X	120
48	48* fragLen = X	168
56	56* fragLen = X	224
64	64* fragLen = X	288

Where fragLen is the length in byte of each uncoded fragment and X is the total length of the data block to be fragmented/defragmented.
The memory requirement is the sum of the “fragment memory store” and “parity matrix” rows.
The data block is reassembled in place directly using the same memory area than the “fragment memory store” , so there is no need for a dedicated “data block” memory space.

For the encoding process, there is no need to store the parity matrix so the required memory simply corresponds to the data block to be segmented and transmitted plus a very little fixed overhead for temporary calculations.

10 Matlab code

Matlab code of the matrix_line function generating a parity check vector:

```
%returns the line N containing M (0 or 1s) of the encoding parity check
matrix
%matrix_line is a vector of 1/0 of size M
function matrix_line = matrix_line(N,M)
matrix_line = zeros(1,M);
s=0;
if (M==32)
    m=32;
else
    m=64;
end
init_v(32)=25;
init_v(40)=7;
init_v(48)=17;
init_v(56)=13;
init_v(64)=2;
x= init_v(M) + 260*N;
nb_coeff=0;
while (nb_coeff<M/2)
    r=255;
    while (r>=M)
        x=prbs16(x);
        r=mod(x, m);
    end
    if (matrix_line(r+1) == 0)
        matrix_line(r+1) = 1; %set to 1 the column which were randomly
selected

    end
    nb_coeff = nb_coeff + 1;
end
```

The prbs16() function implements a PRBS generator with 2^16 period.

```
function r=prbs16(start)
x= start;
lsb = bitand(x,1);
x = floor(x/2);
if (lsb==1)
    x = bitxor (x,hex2dec('B400'));
end
r=x;
```

Encoding process in matlab

```
w = 32; %number of uncoded fragments
fragment_size=8; %size of each fragment in byte
nb_fragment_per_frame = 4; %number of fragment per frame

fprintf('\n a minimum of %d frames will be sent,  payload size %d bytes\n
total broacast size %d
bytes\n',w/nb_fragment_per_frame,nb_fragment_per_frame*fragment_size,w*frag
ment_size);
```

```
DATA = mod([0:w*fragment_size-1],256);


% fragmenting in fragments and frames.


UNCODED_F = zeros(w,fragment_size); %sequence of fragments we want to send
for k=1:w
    UNCODED_F(k,:) =  DATA( (k-1)*fragment_size+1:k*fragment_size);
end

% encode. we will create 3 times w CODED fragments, we may not use them
all. Those can be generated by the TX on the fly one by one.
CODED_F = []; % coded frgament we generate by parity check
for y=1:2*w
    s=0;
    A = matrix_line(y,w);
    for x=1:w

        if (A(x) == 1)
            s = bitxor(s,UNCODED_F(x,:));
        end
    end
    CODED_F = [CODED_F ; s];
end

% create the actual frames , and save a subset of them to disk for future
demodulation
counter = 0;
BROADCAST = {};
clear('tmp');
per = 0.0; %Packet Error Rate

for k=1:2*w/nb_fragment_per_frame
    payload = [];
    for p=1:nb_fragment_per_frame
        counter = counter + 1;
        payload = [payload CODED_F(counter,:)];
    end
    tmp.payload =  payload; %first byte is first frame fragment index ,
second is total nb of fragment , 3rd number of fragment per frame
    tmp.fragment_index = (k-1)*nb_fragment_per_frame;
    tmp.fragment_number = w;
    tmp.fragment_per_frame = nb_fragment_per_frame;

    if (rand() > per) %only save this FRAME with a probability of 1-per
        BROADCAST{end+1} = tmp;
    end
end
```

Decoding process in matlab

```
% recreate the coded fragments
CODED_F = {};
w = BROADCAST{1}.fragment_number;
```

```
nb_fragment_per_frame = BROADCAST{1}.fragment_per_frame;

fragment_size = length(BROADCAST{1}.payload) / nb_fragment_per_frame; %
length of the data portion of the broadcast payload

clear('tmp');


% recover the subset of broadcast frames from the disk file and tabulate
the fragments (a frame may cary several fragments)
for k = 1:length(BROADCAST) %nb of FRAMES
    for p=1:nb_fragment_per_frame
        tmp.fragment = BROADCAST{k}.payload([1+(p-
1)*fragment_size:p*fragment_size]);
        tmp.nb = BROADCAST{k}.fragment_index + p;
        CODED_F{end+1} = tmp;
    end
end

list_of_frag = zeros(w,fragment_size);
B = zeros(w,w);
index = [];


frag_counter = 0;
rec=zeros(1,3*w); %for plot purposes only

for k=1:length(CODED_F) %then analyse the received fragments.

    frag_counter = frag_counter + 1;
    f = CODED_F{k}.fragment;
    f_index = CODED_F{k}.nb;
    rec(f_index) =  1; %for plot only
    line = matrix_line(f_index,w); %line of the matrix used to encode the
received coded frag
    if isempty(index) %if this is the first coded frag received


        index = find(line==1,1,'first'); %record position of the first 1
        list_of_frag(index,:) = f; %store coded frag
        B(index,:)  = line;  % store corresponding coding combination
        index_order = 1;
    else
        % try to eliminate 1s in the received coding line
        for p=1:length(index)
            o =  index_order(p);
            col = index(o); % this is the column of the left most 1's
received up to now
            if (line(col) == 1) % then eliminate by XORING the 2 frag
together
                fprintf('for coded frag %d , eliminating col %d\n',
f_index,col);
                %
disp(line)
                %                                                 disp(B(o,:))
                line = bitxor(line,B(col,:));
                %                                                 disp(line)
```

```
                f = bitxor(f,list_of_frag(col,:)); % and also XOR the coded
fragments in place
            end
        end
        if ~isempty(find(line==1)) %if the line hasn't been reduced
completely then store it as a new fragment
            fprintf('adding frag %d to memory store\n',f_index);


            index = [index find(line==1,1,'first')]; %record position of
the first 1
            list_of_frag(index(end),:) = f; %store coded frag
            B(index(end),:)  = line;  % store corresponding coding
combination
            [tmp,index_order] = sort(index); % order the indexes from
leftmost to right most
            if length(index) == w % matrix rank is now w , we have enough
info to proceed with inversion
                break;
            end
        else
            fprintf('coded frag %d does not bring new information , it was
dropped\n',f_index);
        end
    end

end
fprintf('starting decode after reception of coded fragment %d , %d
fragments were necessary to recover original
content\n',f_index,frag_counter);

figure(1)
clf;
image(B*1000)


% we now have enough to invert and recover uncoded fragments.


% the matrix B is now triangular.  The last line has already a single 1 on
the right most column.
% This means that the fragment list_of_frag(index_order(w)) is already
decoded.
% so we will start from the second line from bottom and remove all 1s
remaining right of the diagonal

for k=w-1:-1:1
    o = index_order(k);
    line = B(k,:);
    f = list_of_frag(k,:);
    for p=k+1:w %eliminate remaining 1's outside diagonal
        if line(p)==1
            list_of_frag(k,:) = bitxor(list_of_frag(k,:), list_of_frag(p,:)
); % we process fragments in place in the memory , no need to duplicate
        end
    end
end
```

```
DATA_RECONSTRUCTED =  reshape(list_of_frag',1,w*fragment_size);
```

11 Table of possible data block length

total block size in bytes	nb of fragments	fragment size in byte
32	32	1
40	40	1
48	48	1
56	56	1
64	64	1
80	40	2
96	48	2
112	56	2
120	40	3
128	64	2
144	48	3
160	40	4
168	56	3
192	64	3
200	40	5
224	56	4
240	48	5
256	64	4
280	56	5
288	48	6
320	64	5
336	56	6
352	32	11
360	40	9
384	64	6
392	56	7
400	40	10
416	32	13
432	48	9
440	40	11
448	64	7
480	48	10
504	56	9
512	64	8
680	40	17
816	48	17
832	64	13
840	56	15
864	48	18
880	40	22
896	64	14
912	48	19
920	40	23
928	32	29
952	56	17
960	64	15
992	32	31
1000	40	25
1008	56	18
1024	64	16
1040	40	26
1056	48	22
1064	56	19
1080	40	27
1088	64	17
1104	48	23
1120	56	20
1152	64	18
1160	40	29
1176	56	21
1184	32	37
1200	48	25

Block sizes bigger than 1200 bytes are possible , max block size is 3072 bytes assuming 50 bytes of application payload pre frame (64 x 48 bytes fragments)