LoRaWAN Fragmented Data Block Transport Specification v1.0.0

NOTICE OF USE AND DISCLOSURE

LoRa Alliance™
5177 Brandin Court
Fremont, CA 94538
United States

LoRaWAN Fragmented Data Block Transport Specification

Authored by the FUOTA Working Group of the LoRa Alliance Technical Committee

Technical Committee Chairs:
N.SORNIN (Semtech), A.YEGIN (Actility)

Working Group Chairs:
J.CATALANO (Kerlink), N.SORNIN (Semtech)

Editor:
J.CATALANO (Kerlink)

Contributors:
J.CATALANO (Kerlink), J-P.COUPIGNY (STMicroelectronics), N.SORNIN (Semtech), J.STOKKING (The Things Network Foundation)

Version: v1.0.0
Date: September 10, 2018
Status: Final release

Contents

Tables

Figures

1 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

The octet order over the air for all multi-octet fields is little endian (Least significant byte is sent first).

2 Introduction

This document proposes an application layer messaging package running over LoRaWAN to perform the following operations on a fleet of end-devices:

- Send a fragmented block of data to one or many end-devices

All messages described in this document are transported as application layer messages. As such, all unicast messages (uplink or downlink) are encrypted by the LoRaWAN MAC layer using the end-device's AppSKey. Downlink multicast messages are encrypted using a multicast group McAppSKey common to all end-devices of the group. The setup of the group is described in [RPD_Remote_Multicast_Setup].

The data block transported may be a firmware upgrade, but this document is not specific to firmware upgrade. Any large (from 1kBytes to X Kbytes) data file may be sent to a (group of) end-device using this protocol.

The "**fragmentation control**" package can be used to:

- Setup / report / delete fragmentation transport sessions
- Several fragmentation sessions MAY be supported simultaneously by an end-device
- Fragmentation can be used either over multicast or unicast
- Authenticate a data block once reconstructed (TBD)
- Report on the status of a fragmentation session

This package uses a dedicated port to separate its traffic from the rest of the applicative traffic.

3 Downlink Fragmentation Transport Message Package

The identifier of the fragmentation transport package is 3. The version of this package is version 1.

This package supports all the commands necessary to transport reliably a large data block from a fragmentation server to an end-device (using unicast) or a group of end-device (if multicast is used over classB or classC). This package requires a dedicated port. The default port value is 201. Once declared, this port cannot be used for any other purposes.

All fragmentation related messages are exchanged on this port using application payload and encrypted using the end-device's AppSKey or the McAppSKey. All unicast or multicast control messages use the same format:

Command1	Command1 Payload	Command2	Command2 payload	….

A message MAY carry more than one command with the exception of the "Data fragment" command which MUST be the only command in a message's payload. The length of each command's payload is fixed and a function of the command. Commands are executed from first to last.

The following table summarizes the list of fragmentation control messages

CID	Command name	Transmitted by		Multicast (M) / Unicast (U)	Short Description
		End-device	server		
0x00	***PackageVersionReq***		x	U	Used by the AS to request the package version implemented by the end-device
0x00	***PackageVersionAns***	x		U	Conveys the answer to PackageVersionReq
0x01	***FragStatusReq***		x	U/M	Asks an end-device or a group of end-devices to send the status of a fragmentation session
0x01	***FragStatusAns***	x		U	Conveys answer to the FragSessionStatus request
0x02	***FragSessionSetupReq***		x	U	Defines a fragmentation session
0x02	***FragSessionSetupAns***	x		U	
0x03	***FragSessionDeleteReq***		x	U	Used to delete a fragmentation session
0x03	***FragSessionDeleteAns***	x		U	
0x08	***DataFragment***		x	U/M	Carries a fragment of a data block

Table 1: Fragmentation Control messages summary

The message marked "U/M" can be received using a unicast or multicast address. All other messages are exchanged only using the unicast end-device address.

3.1 PackageVersionReq & Ans

The ***PackageVersionReq*** command has no payload.
The end-device answers with a ***PackageVersionAns*** command with the following payload.

Field	PackageIdentifier	PackageVersion
Size (bytes)	1	1

Table 2: PackageVersionAns

PackageIdentifier uniquely identifies the package. For the “fragmentation transport package” this identifier is 3.
PackageVersion corresponds to the version of the package specification implemented by the end-device.

3.2 FragSessionSetupReq & Ans

This message is used to setup a DL fragmentation transport session.

Field	FragSession	NbFrag	FragSize	Control	Padding	Descriptor
Size (bytes)	1	2	1	1	1	4

Table 3: FragSessionSetupReq

FragSession identifies the fragmentation session and contains the following fields

FragSession Fields	RFU	FragIndex	McGroupBitMask
Size (bits)	2bits	2bits	4bits

Table 4: FragSessionSetupReq FragSession field

FragIndex [0 to 3] identifies one of the 4 simultaneously possible fragmentation sessions.

McGroupBitMask specifies which multicast group addresses are allowed as input to this defragmentation session. Bit number X indicates if multicast group with McGroupID=X is allowed to feed fragments to the defragmentation session. Unicast can always be used as a source for the defragmentation session and cannot be disabled. For example, 4’b0000 means that only Unicast can be used with this fragmentation session. 4’b0001 means the defragmentation layer MAY receive packets from the multicast group with McGroupID=0 and the unicast address. 4’b1111 means that any of the 4 multicast groups or unicast may be used. If the end-device does not support multicast, this field SHALL be ignored.

> **Note**: the McGroupBitMask is a mechanism allowing tying a defragmentation session to one or several specific multicast group addresses in a given end-device. For example, a street lighting controller end-device is part of 2 multicast groups, one used to control the lamps and one for firmware updates. Only data fragments coming from the second group shall be taken into account by the fragmented transport layer. The first group shall only transport ON/OFF lamp control packet and should not be allowed to transport firmware update data.

NbFrag (Number of Fragments) specifies the total number of fragments of the data block to be transported during the coming multicast fragmentation session. (example: 100 means that the data block that is going to be multicasted will be divided in 100 fragments)

FragSize (fragment size) is the size in byte of each fragment. The data block size is therefore NbFrag x FragSize

Control consists of 2 fields

Control Fields	RFU	FragAlgo	BlockAckDelay
Size (bits)	2bits	3bits	3bits

Table 5: FragSessionSetupReq Control field

FragAlgo encodes the type of fragmentation algorithm used. This parameter is simply passed to the fragmentation algorithm. FragAlgo 0 corresponds to FEC fragmentation described in Appendix "data block fragmentation forward error correction code proposal".

BlockAckDelay encodes the amplitude of the random delay that end-devices have to wait between the reception of a downlink command sent using multicast and the transmission of their answer. This parameter is a function of the group size and the geographic spread and is used to avoid too many collisions on the uplink due to many end-devices simultaneously answering the same command. The actual delay SHALL be $rand().2^{BlockAckDelay+4}$ seconds where $rand()$ is a random number in the [0:1] interval.

Padding: The binary data block size may not be a multiple of FragSize. Therefore, some padding bytes MUST be added to fill the last fragment. This field encodes the number of padding byte used. Once the data block has been reconstructed by the receiver, it SHALL remove the last "*padding*" bytes in order to get the original binary file.

Descriptor: The descriptor field is a freely allocated 4 bytes field describing the file that is going to be transported through the fragmentation session. For example, this field MAY be used by the end-device to decide where to store the defragmented file, how to treat it once received, etc... If the file transported is a FUOTA binary image, this field might encode the version of the firmware transported to allow end-device side compatibility verifications. The encoding of this field is application specific.

The end-device answers with a **FragSessionSetupAns** message with the following payload

FragSessionSetupAns payload	StatusBitMask
Size (bytes)	1

Table 6: FragSessionSetupAns

Bits	7:6	5:4	3	2	1	0
Status bits	FragIndex	RFU	Wrong Descriptor	FragSession index not supported	Not enough Memory	Encoding unsupported

Table 7: FragSessionSetupAns StatusBitMask field

If any of the bits [0:3] is set to 1 the **FragSessionSetup** command was not accepted.

If a FragSessionSetupReq command with a FragIndex field corresponding to an already existing fragmentation session is received, the context of this session is cleared, and a new session is setup with the parameters of the new FragSesisonSetupReq command.

3.3 FragSessionDeleteReq & Ans

This message is used to delete a fragmentation session. A fragmentation session MUST be deleted before its index (FragIndex) can be reused for another one. The command payload is:

Field	Param
Size (bytes)	1

Table 8: FragSessionDeleteReq

Where:

Bits	7:2	1:0
Param bits	RFU	FragIndex

Table 9: FragSessionDeleteReq Param bits

The end-device answers with FragSessionDeleteAns with payload:

Field	Status
Size (bytes)	1

Table 10: FragSessionDeleteAns

Where:

Bits	7:3	2	1:0
Status bits	RFU	Session does not exist	FragIndex

Table 11: FragSessionDeleteAns Status bits

If the bit 2 is set to 1 the **FragSessionDeleteReq** command was not accepted because the fragmentation session corresponding to FragIndex did not exist in the end-device.

3.4 Downlink Data Fragment message

This message can be received by the end-device in a multicast or unicast downlink frame. This message is used to carry a data block fragment.

The payload content is:

Size (bytes)	Index&N	0:MaxAppPl-3
Payload	2	P_M^N

Table 12: Downlink Data Fragment payload

Index&N Fields	FragIndex	N
Size (bits)	2bits	14bits

Table 13: Downlink Data Fragment Index&N field

If this message was received on a multicast address, the end-device MUST check that the multicast address used was enabled at the creation of the fragmentation session through the *McGroupBitMask* field of the ***FragSessionSetup*** command. If not, the frame SHALL be silently dropped.

Where P_M^N is the fragment N over M of the session.

More than M fragments MAY actually be transmitted to add redundancy and packet loss robustness. N is the index of the coded fragment transported.

M is equal to the *NbFrag* parameter.

Once the data block has been reconstructed the end-device SHALL drop any further message using that *fragIndex* until the fragmentation session is first deleted and a new fragmentation session is setup through the ***FragSessionSetup*** application command.

3.5 FragSessionStatusReq & Ans

This message can be transmitted by the server in a UNICAST or MULTICAST downlink frame.

Size (bytes)	1
FragParam Payload	FragStatusReqParam

Table 14: FragSessionStatusReq

Where:

bits	7:3	2:1	0
FragStatusReqParam field	RFU	FragIndex	Participants

Table 15: FragSessionStatusReq FragStatusReqParam field

Used by the fragmentation server to request receiver end-devices to report their current defragmentation status.

The receivers (in the case of multicast) 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 responses as specified by the BlockAckDelay field of the FragSessionSetupReq command.

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

Table 16: FragSessionStatusReq FragStatusReqParam field participant bit

The end-devices respond with a ***FragSessionStatusAns*** message***.*** The message payload is:

Size (bytes)	2	1	1
FragParam Payload	Received&index	MissingFrag	Status

Table 17: FragSessionStatusAns

Where:

bits	15:14	13:0
Received&index field	FragIndex	NbFragReceived

Table 18: FragSessionStatusAns Received&index field

bits	7:1	0
Status field	RFU	Not enough matrix memory. The defragmentation process was aborted because the number of missing fragments was greater than the available memory matrix storage capacity

Table 19: FragSessionStatusAns Status field

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

NbFragReceived is the total number of fragments received for this fragmentation session since the session was created.

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. If more than 255 fragments are missing, then MissingFrag SHALL be set to 255.

As described in the "***FragSessionStatusReq***" command, the receivers MUST respond with a pseudo-random delay as specified by the BlockAckDelay field of the FragSessionSetupReq command.

4 File integrity check and authentication

The payloads transported by the fragmentation/defragmentation package are encrypted and authenticated using the McAppSKey and McNwkSKey. However, those keys are identical in all the end-devices of the multicast group. Because one of the group's end-devices might be compromised (the end-device might have been physically destroyed and the keys extracted), those keys cannot be considered safe except if a tamper-proof secure element is used to store them in ALL the end-devices of the group.
If that is not the case (no secure element is used), then an additional file integrity and authentication step SHOULD take place. The integrity/authentication check corresponds to making sure that the block reconstructed is exactly what the fragmentation server wanted to send to the end-device and that this block has not been modified in any way through the transport process. This goal may be achieved by different means:

1. Public/private cryptography certificate
2. Unicast exchange protected by Symmetric key

Solution 1 does not require any additional exchange and MAY use a standard HASH + certificate mechanism based on RSA or ECC cryptography. This solution is RECOMMENDED when the fragmentation layer is used to transport a firmware upgrade file. That solution increases the size of the file transported (cryptography/certificate overhead). For ECC that overhead is typically around 100bytes.
Solution 2 relies on a unicast exchange between the end-device and the fragmentation server. The messages exchanged contain a HASH of the reconstructed file and MUST be protected by an end-device specific key. In that way even if the multicast keys are considered unsafe, the final authentication is made on an end-device per end-device basis and cannot be compromised. That solution has no file size overhead but requires an additional unicast exchange between each end-device and the AS.
The choice between the two solutions is application specific and this is currently considered out-of-scope of this specification. Following versions of this specification will provide a recommended file integrity/authentication verification process.

5 Fragmentation algorithm

This section will contain a description of the fragmentation / coding / decoding / defragmentation algorithm proposed.

The coding adds some redundancy to the fragment transmitted such that an end-device missing some of the multicasted fragments can still reconstruct the complete data block.

The maximum ratio of lost fragment that can be tolerated is a parameter selected by the "fragmentation server" preparing the multicast.

APPENDIX: DATA BLOCK FRAGMENTATION FORWARD ERROR CORRECTION CODE PROPOSAL

6 Introduction

This appendix proposes a simple Forward Error Correction (FEC) code to be used for fragmented transport of large binary files over LoRaWAN. As all radio link, a LoRaWAN link exhibit a certain ratio of lost frames. Adding FEC in the file fragmentation process allows an end-device to autonomously recover the full file even in the presence of lost frames without having to systematically request the missing fragments.
The transmitter of the fragmented binary file can select to add an arbitrary redundancy to the transmission content through this FEC.
For example a 10% redundancy added by the fragmentation transmitter allows the receiver performing the defragmentation to loose roughly 10% of the incoming frames and still be able to reconstruct the binary file.

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 end-device receives a notification and stops transmitting.

The coding scheme proposed here 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

7 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 + 2 bytes

Where fragLen is the byte length of each fragment plus 2 bytes of fragmentation header (containing Index & N, see 3.4)

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. The coded fragments $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ all have exactly the same byte length than the uncoded (Bn) fragments.
To allow the original uncoded fragments reconstruction on the receiving end of the link even in presence of arbitrary packet loss, the transmitter (performing the fragmentation) 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 (or redundancy factor) 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 :

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

Where $C_M^N(i)$ is a function of M,N and i and whose value is either 0 or 1.

- $0.B_1$ is a binary 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 $C(N,M) = [C_M^N(1), C_M^N(2), \dots, C_M^N(M)]$ 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 either:

- If N<=M , a vector of length M with a single one at position N, all other bits = 0
- If N>M : 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=26 and with a coding ratio CR=1/2 , this matrix has 26/CR = 52 lines. Id, it allows the creation of 52 coded fragments from the 26 original uncoded fragments.

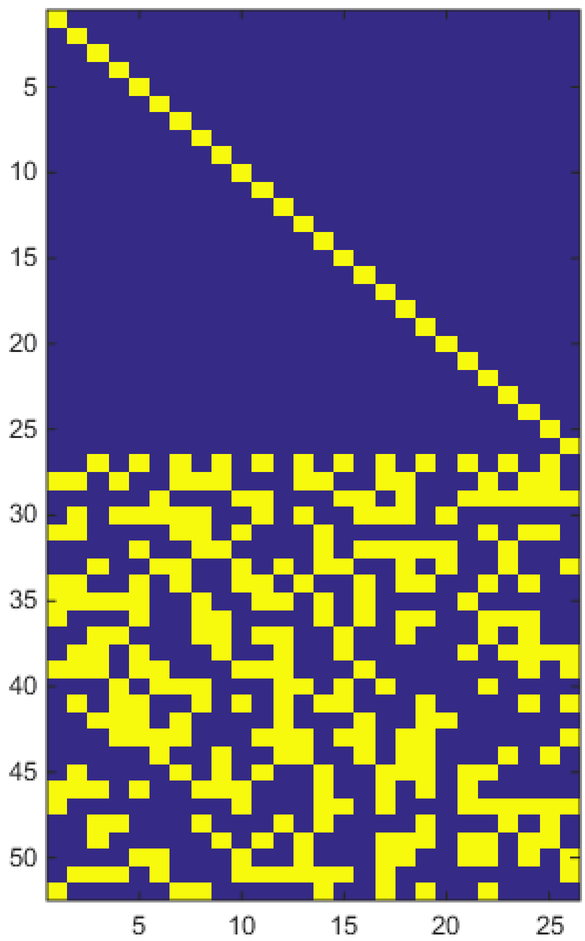


Figure 1 : 26x52 parity check matrix

We can see that the parity matrix consists first of an MxM identity matrix followed by a parity control MxM matrix.

The coded fragment $\mathrm{P}_{\mathrm{M}}^{\mathrm{N}}$ is therefore the bit-wise Xor of the uncoded fragment B_i such that $C_M^N(i)$ is non-zero. The coded fragments have exactly the same bit length than the uncoded fragments.

<u>Step by Step encoding example:</u>

The transmitter must send 2000 bytes allowing up to 50% packet error rate on the radio link. A coding ratio of ½ is selected. For this purpose the 2000 bytes data block will be segmented in 100 fragments of 20 bytes each , each frame will transport 1 fragment. The transmitter will send 100/CR = 200 coded fragments. We will see that the receiver will be able to decode as soon as it receives ~103 frames out of the 200 (depending on the exact combination of frames lost).

First split the 2000 bytes into 100 uncoded fragments of 20 bytes each, B1 to B100.

To generate the first coded fragment.
First generate the first line of the parity check matrix by calling C = matrix_line(1,100)
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'b1 followed by 99 zeros, the first coded fragment P_{100}^{1} = B1

When the transmitter reaches the frame number 101 , we have for example:
C= matrix_line(101,100) = 100'b0110010………;
Therefore P_{100}^{101} = B2 + B3 +B6 + ……;
Where + is the bit-wise Xor operator

8 Fragment decoding and reassembling

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

For each frame received extract the coded fragment and its index.

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

Then process those fragments one by one.

1. For each new fragment $\mathrm{P_M^N}$, 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 $\mathrm{P_M^N}$ and the coded fragment stored at position i in the fragment memory store S_i and update $\mathrm{P_M^N}$ 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 $\mathrm{P_M^N}$; 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 $\mathrm{P_M^N}$ 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 $S_1 : S_2 : ... : S_M$ where : represents the concatenation operator.

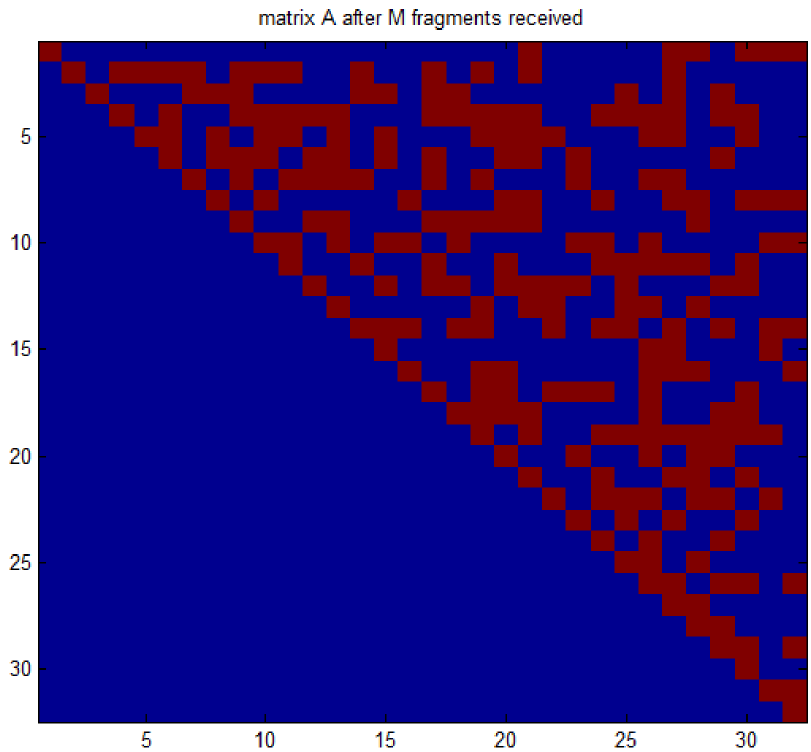
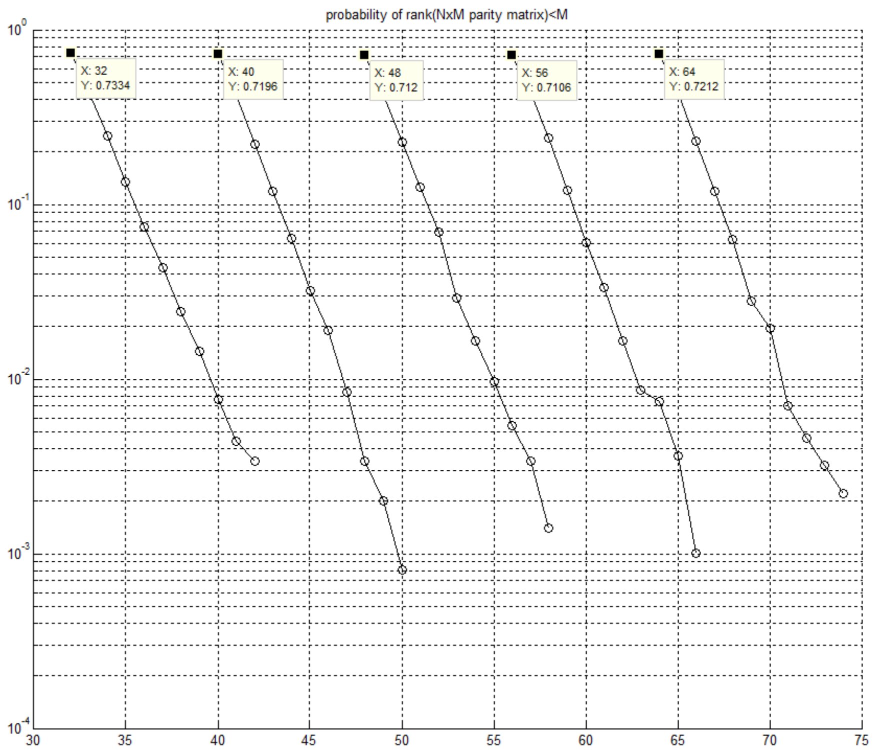


Figure 2 : 32x32 matrix A built during decoding process

9 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 fragments. This coding scheme with a fixed overhead is therefore close to ideal performance when the number of fragment is large (>100) , because adding 2 fragments on a 100 fragments session only represent 2% relative statistical overhead. The overhead increases when the number of fragments is lower. This coding scheme should not be used for less than 20 fragments (the average overhead is 10% in that case).

10 End-device memory requirement

This coding scheme has been optimized to allow the transportation of large binary file with minimum memory overhead on the receiving end (performing the defragmentation).
The following formula gives the decoding memory requirements for an optimized end-device implementation expressed in **Bytes** on **TOP** of the memory required to store the final reconstructed data block.

> **Example**: a 50kbytes data block must be received. The end-device requires 50kbytes of available memory + the value given by the following formula

For a data block fragmented using M fragments, let L be the number of coded fragments **lost** by the end-device amongst the first M coded fragments. (The fragments $\mathrm{P}_{\mathrm{M}}^{1}$ to $\mathrm{P}_{\mathrm{M}}^{\mathrm{M}}$).

The required defragmentation memory overhead is a function of the maximum L value that the end-device is designed to tolerate. Interestingly, the memory overhead is **NOT** a function of the total number of fragments used to convey the data block. This is because the defragmentation implementation is optimized to use the fact that the first M fragments actually contain the original uncoded data block ($\mathrm{P}_{\mathrm{M}}^{\mathrm{i}} = \mathrm{B}_{\mathrm{M}}^{\mathrm{i}}\ for\ i \leq M$).

Parity Matrix memory (bytes) = *l*.(*l*+1)/2/8 + 2.*l* , where *l* is the maximum value of L that the end-device tolerates (if L>*l* , the defragmentation fails and is aborted)

The following table gives a few numerical examples.

l	Parity matrix memory
32	130
40	183
48	243
56	312
64	388

The total memory requirement is the sum of the full data block size and the parity matrix memory size.

> Example: a 50kbytes data block must be received, as 1000x50bytes fragments. The end-device is designed to be able to withstand the loss of 64 fragments out of the first 1000 transmitted (*l*=64). The end-device requires 50kbytes of available memory + 388bytes to store the parity matrix

The data block can be reassembled in the memory directly at its final address (this means directly in the FLASH memory space for most end-devices), so there is no need for an additional “data block” swap memory space.
The parity matrix memory space can be erased and reused once the defragmentation is finished.

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.

11 Preliminary Matlab code

This matlab code generates a list of coded fragments from an arbitrary binary file

Notations:

```
w = 32; %the number of fragments into which the binary file is split
fragment_size=10; %the size of each fragment in bytes
DATA; %a vector of bytes = the binary file to be sent. The length must be
w*fragment_size
```

Encoding process in matlab

```
w = 32; %number of uncoded fragments
fragment_size=10; %nb of bytes per fragment
fprintf('\n number of uncoded fragments:%d,  fragment size (bytes):%d bytes\n total
broacast size %d bytes\n',w,fragment_size,w*fragment_size);

DATA = mod([0:w*fragment_size-1],256); %arbitrary binary file to be sent

% start of the fragmentation and encoding process
% UNCODED_F is an array of uncoded fragments
UNCODED_F = zeros(w,fragment_size);
for k=1:w
    UNCODED_F(k,:) =  DATA( (k-1)*fragment_size+1:k*fragment_size);
end


% now encode.
% we will create 2w CODED fragments, this number is arbitrary. Those can be
generated by the transmitter on the fly one by one.
CODED_F = []; % this will contain the array of 2w coded fragments
for y=1:w
CODED_F(y,:) = UNCODED_F(y,:); %the first w coded fragments are equal to the
uncoded fragments (binary data without coding)
end
%then we add w parity check fragments
for y=1:w
    s=zeros(1,fragment_size);
    A = matrix_line(y,w); %line y of w.w matrix
    for x=1:w
        if (A(x) == 1) %if bit x is set to 1 then xor the corresponding fragment
            s = bitxor(s,UNCODED_F(x,:));
        end
    end
    CODED_F = [CODED_F ; s]; % add the resulting coded fragment to the list
end
```

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

```
%this funciton returns line N of the MxM parity matrix
function matrix_line = matrix_line(N,M)
matrix_line = zeros(1,M);
s=0;

% we must treat powers of 2 differently to make sure mtrix content is close
% to random . Powers of 2 tend to generate patterns
if (M == 2^floor(log2(M))) % if M is a power of 2
```



```
    m=1;
else
    m=0;
end


x= 1+1001*N; %initialize the seed differently for each line
nb_coeff=0;
while (nb_coeff<floor(M/2)) % will generate a line with M/2 bits set to 1 (50%)
    r=2^16;
    while (r>=M) %this can happen if m=1, in that case just try again with a
different random number
        x=prbs23(x);
        r=mod(x, M+m);  %bit number r of the current line will be switched to 1
    end

    matrix_line(r+1) = 1; %set to 1 the column which was randomly selected
    nb_coeff = nb_coeff + 1;
end
```

The prbs23() function implements a PRBS generator with 2^23 period.

```
%standard implementation of a 23bit prbs generator
function r=prbs23(start)
x= start;
b0 = bitand(x,1);
b1 = bitand(x,32)/32;
x = floor(x/2) + bitxor(b0,b1)*2^22;
r=x;
```

12 Glossary

AES	Advanced Encryption Standard
AS	Application Server
FEC	Forward Error Correction
TBD	To Be Done

13 Bibliography

13.1 References

[LoRaWAN 1.0.2]: LoRaWAN™ 1.0.2 Specification, LoRa Alliance, July 2016

[LoRaWAN 1.1]: LoRaWAN™ 1.1 Specification, LoRa Alliance, October 11, 2017

[LoRaWAN_Remote_Multicast_Setup]: LoRaWAN Remote Multicast Setup Specification v1.0.0, LoRa Alliance, September 10, 2018

14 NOTICE OF USE AND DISCLOSURE

The information within this document is the property of the LoRa Alliance ("The Alliance") and its use and disclosure are subject to LoRa Alliance Corporate Bylaws, Intellectual Property Rights (IPR) Policy and Membership Agreements.

Elements of LoRa Alliance specifications may be subject to third party intellectual property rights, including without limitation, patent, copyright or trademark rights (such a third party may or may not be a member of LoRa Alliance). The Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

This document and the information contained herein are provided on an "AS IS" basis and THE ALLIANCE DISCLAIMS ALL WARRANTIES EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO (A) ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OF THIRD PARTIES (INCLUDING WITHOUT LIMITATION ANY INTELLECTUAL PROPERTY RIGHTS INCLUDING PATENT, COPYRIGHT OR TRADEMARK RIGHTS) OR (B) ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NONINFRINGEMENT.

IN NO EVENT WILL THE ALLIANCE BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OF DATA, INTERRUPTION OFBUSINESS, OR FOR ANY OTHER DIRECT, INDIRECT, SPECIAL OR EXEMPLARY, INCIDENTIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES OF ANY KIND, IN CONTRACT OR IN TORT, IN CONNECTION WITH THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The above notice and this paragraph must be included on all copies of this document that are made.

LoRa Alliance™
5177 Brandin Court
Fremont, CA 94538
United States

Note: All Company, brand and product names may be trademarks that are the sole property of their respective owners.