Multimedia Processing for Streaming Applications

Luby transform codes

Luigi Schiavone	matr. 062270**1522**
Anna D'Andrea	matr. 062270**1682**
Carmine Coppola	matr. 062270**1699**
Pierpaolo Della Monica	matr. 062270**1701**
Enrico Maria Di Mauro	matr. 062270**1706**
Allegra Cuzzocrea	matr. 062270**1707**

Summary

Source coding

Source coding deals with how to compress a source of information with the goal of achieving **the most compact representation possible**

In this case we don't consider the transmission channel but only the encoder that takes care of the coding

Source coding **eliminates redundancies from a source**

Examples of **lossless source coding** algorithms are:

- Huffman
- Lempel-Ziv

An example of **lossy source coding** algorithms is:

- Lloyd-Max

Channel coding

The **channel** through which an information source is transmitted **introduces errors**

Channel coding combats their effect by using a **controlled redundancy**. Errors can be detected or even corrected

Redundancy consists of **adding** some **information** to the source

A simple kind of redundancy is the **repetition of binary digits**. The receiver decides the correct ones **by majority**

A more complex kind of redundancy is used by **Hamming codes**. The receiver decides the correct binary digits through **parity checks**

Channel coding

What matters in channel coding is the **minimum distance decision**

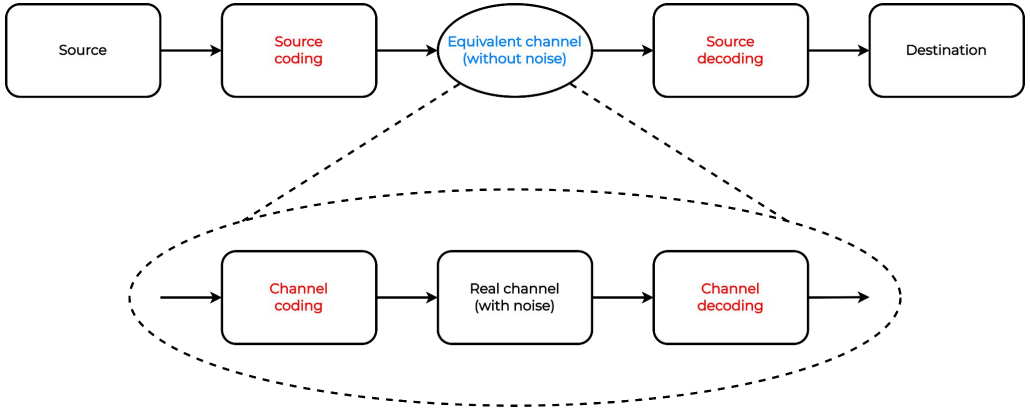
Consider the following example: the decoder receives the triplet 010 and **it decides at the minimum distance.**
010 is closer to 000

If just 1 of 3 binary digits is wrong it gets the right answer.
However, if 2 of 3, or more, are wrong it gets the wrong answer

It is a minimum distance decoding. Regarding Hamming codes, the relevant distance is the Hamming distance, not the Euclidean distance

Source coding vs channel coding

The **goal** of the **channel coding** is to **increase** the **resistance** of a communication system to the noise present on the channel



Channel coding **transforms input** source into a **new source** that is **more robust** to the effects of noise. Then, channel **decoding** performs the **reverse operation** and it **reconstructs** the original **source**

Binary erasure channel

Errors are **introduced** into the streams of transmitted information **since** the **real channels** are **noisy** and interfering

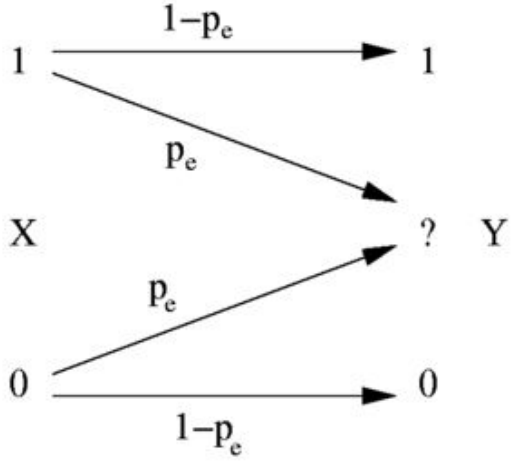
In coding theory and information theory a **BEC** (Binary Eraser Channel) is a **communications channel** model that **represent** this **behavior**

The binary erasure channel of communication was introduced by Elias in 1955, but it was regarded as a rather theoretical channel model until the large-scale deployment of the Internet about 40 years later

Binary erasure channel

A BEC is a **communications channel** model, that **represent** this **behavior**

A transmitter **sends** a **bit** and the receiver with some probability p_e receives a **message** that the **bit** was not received ("erased") and with probability $1 - p_e$ receives the bit **correctly**



The binary erasure channel has:

- 2 inputs: 0, 1
- 3 outputs: 0, 1, error (i.e. something unreadable)

Real world model of the BEC

On the Internet, data is transmitted in the form of **packets** that are routed on the network from the sender to the receiver. Due to various reasons, for example buffer overflows at the intermediate routers, **some packets may get lost** and never reach their destination. Other packets may be declared as lost if the internal checksum of the packet doesn't match

Therefore, **Internet** is a very good real-word model of the **BEC**

Raptor Codes, IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 52, NO. 6, JUNE 2006

Erasure codes

Erasure codes are a type of **encoding** of **data**

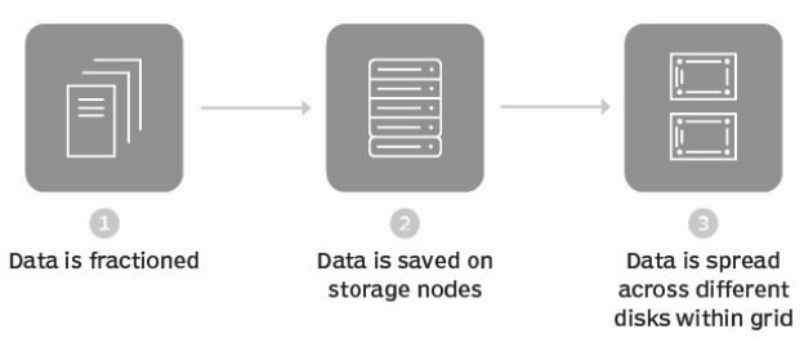
This are **codes** which, under the **assumption** of **transmission** on a **BEC**, **transform** a **message** of symbols into a **longer message** of codewords, such that the **original message** can be **recovered** from a **subset** of the **codewords**

Erasure coding is a method of **data protection** in which data is **broken** into **fragments**, **expanded** and **encoded** with redundant data pieces and **stored** across a set of **different locations** or storage media

Erasure codes

The following is an **example** of the use for this **encoding**

If a drive fails or data becomes corrupted, **the data can be reconstructed** from the segments stored on the other drives. In this way, **erasure coding can help increase data redundancy** without the overhead or limitations that come with different implementations of RAID (Redundant Array of Independent Disks)



Erasure coding makes it possible to protect data without having to fully replicate it because **the data can be reconstructed** from parity fragments

Erasure codes

They are especially **useful** for **transmission** of data across a **network that can drop packets of data**, when it is **impractical** for the **receiver** to be in **constant communication** with the sender

Given a file with k blocks, the sender generates $B > k$ encoded blocks. The code is designed so that after receiving any set of at least K blocks, for some K slightly larger than k, **the original data can be decoded with high probability**

For scenarios like broadcast or for networks with very long one-way delays (think the Mars rover sending an image to Earth), this is much more practical than the receiver acknowledging every block, as done in the TCP (Transmission Control Protocol), which is used on the web and by many other internet applications

What is a rate?

Given k **symbols** of message and K message of **codewords**, the fraction $r = k/K$ is called the **code rate**. The **symbol length** for the codes can be **arbitrary**, from one-bit binary symbols to general l-bit symbols

Rateless codes

Rateless codes are **codes** for which the **number** of **encoding symbols** that can be generated from the data is **potentially limitless,** means that **the transmitter transmits continuously**. It is the basic concept of streaming

Introduction to Luby transform codes

LT codes (Luby Transform codes) were invented by **Michael Luby** in 1998 and published in 2002

The distinctive feature of LT codes is in employing a particularly simple algorithm based on the exclusive or operation $\oplus$ to encode and decode the messages

Introduction to Luby transform codes

LT codes are **rateless** because the **encoding algorithm** can, in principle, **produce** an **infinite** number of message **packets**

LT codes are part of **erasure codes**, so they are erasure correcting codes because they **can** be used to **transmit digital data** reliably **on** an **erasure channel**

Introduction to Luby transform codes

Encoding symbols can be generated on the fly, as few or as many as needed. This is called **digital fountain approach**

The **LT** decoder can **recover,** with high probability, an **exact copy of the data** from any **set** of suitable cardinality of the generated **encoding symbols**. Since the decoder can recover the data from **nearly the minimal number of encoding symbols possible**, this implies that **LT codes are near optimal** with respect to any erasure channel

Furthermore, the encoding and decoding times are **asymptotically very efficient with respect to data length.**

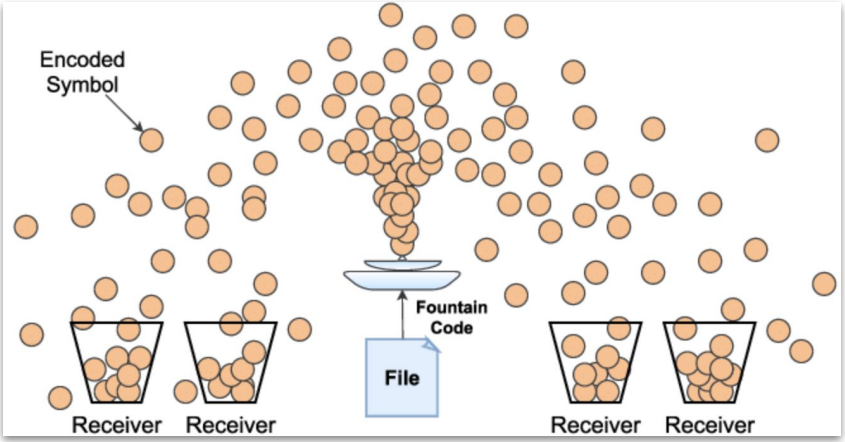
Introduction to Luby transform codes

The evolution of LT codes consists of **Raptor codes**, which have **linear time encoding** and **decoding**. Raptor codes are fundamentally **based** on **LT codes**

Raptor codes are **faster than LT codes** and generally **require fewer blocks** to **decode**. They are **used** in **several communication standards** such as in **broadcast** of video to mobile devices. Their implementation, however, is more complicated

Fountain Codes

The Fountain codes do not have the requirement of the receiver to inform the transmitter of the identity of the packets that are corrupted resulting in **expelling** the **need** of **automatic repeat request transmission**



These codes are called fountain codes, **in an analogy to a constant stream of water** from a fountain

Any **set** of **drops** from the **fountain** will serve the purpose of **filling** the **receiver's bucket**

Fountain Codes

Fountain codes differ from standard channel codes that are characterized by a rate, which is defined at the time of design

The fountain encoder generates a random number of coded symbols by simple arithmetic calculations

Note that more loss of symbols just translates to a longer waiting time to receive the packets

Fountain Codes

Fountain codes are primarily **introduced** for a possible **solution** to **address** the **information** delivery in **broadcast** and **multicast scenarios**; later many more field applications were found such as data storage

The transmitter resembles a fountain emitting encoded symbols until all the interested sinks have collected the minimum number of symbols necessary for decoding

The first class of Fountain codes was invented by Luby. The codes in this class are called LT-codes

How LT codes work

- Encode algorithm
- Decode algorithm

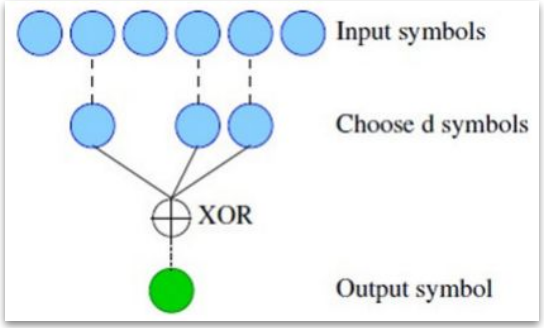
LT encoding algorithm

The data of length N is **partitioned** into $k=N/l$ **input symbols**, where l is the **number** of **bits** per **symbol**. The **process** of generating an **encoding** symbol is conceptually very easy to **describe**:

1. randomly choose the **degree** $\mathbf{d}$ of the encoding symbol from a **degree distribution**
2. choose uniformly at **random** $\mathbf{d}$ **distinct input symbols** as neighbors of the encoding symbol
3. the value of the encoding symbol is the **exclusive-or** of the $\mathbf{d}$ neighbors

Note that the “neighbors” are the blocks of the original source that are chosen uniformly and are XORed together to produce an encoded symbol. Furthermore, bitwise sum modulo 2 of the d input symbols is equivalent to the bitwise XOR operation, denoted as $\oplus$

Encoding process of LT codes



The **quantity** of **output symbols** decided **previously** or when the **receiver** had received **enough output symbols** required for **decoding** of original message can be used as **stopping condition** for encoding algorithm

Note that irrespective of symbol length an input symbol could be a vector of bits or just one bit, regardless of this encoding and decoding process remain same and bitwise XORing of the entire vector is done

LT decoding algorithm

The encoding process requires the receiver to obtain both the **XORing of the information** relating to the source symbols and the **set of neighbors** used

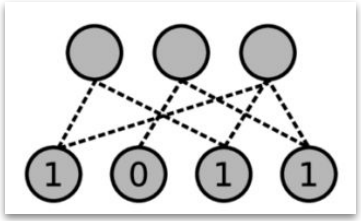
There are many **different ways** of **communicating** this **information** to the **decoder**, depending on the application

Given an **ensemble** of **encoding symbols** and some **representation** of their associated **degrees** and **sets** of **neighbors**, the decoder repeatedly **recovers** input symbols using a **recovery rule**

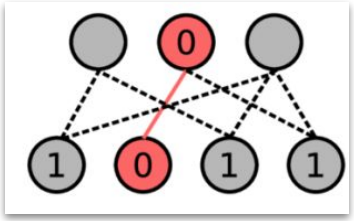
LT decoding algorithm

1. If there is at least one encoding symbol that has **exactly one neighbor**, then the neighbor can be recovered immediately since it is a **copy of the encoding symbol**
2. The value of the recovered input symbol is **XORed** into any **remaining encoding symbols** that also **have** that **input symbol** as a **neighbor**
3. The recovered input symbol is removed as a neighbor from each of these encoding symbols and **the degree of each such encoding symbol is decreased by one** to reflect this removal

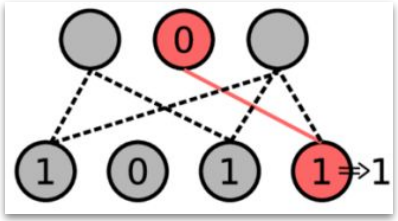
Graphical example of the decoding algorithm



The encoding process build a **graph** characterized by **two** types of **nodes**, the **original ones** and the **encoded ones**



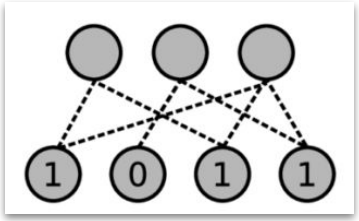
The decoding process begins after the reception of output symbol of **degree one**. This symbol is the exact replica of input symbol

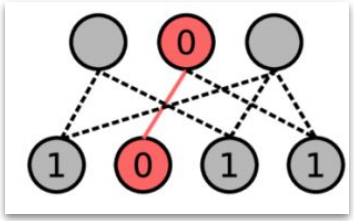


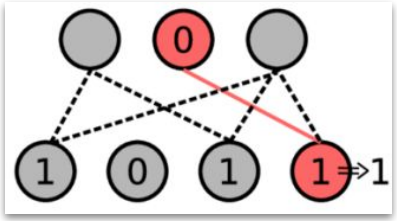
So we have one **uncovered input data symbol**. Then, the algorithm performs the **XOR operation** to all its neighbouring symbols **removing the edges** in the graph between recovered input data symbols and their neighbouring symbols

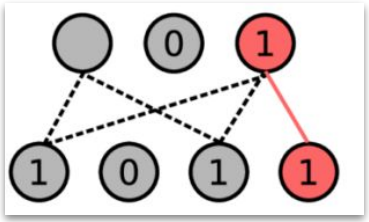
Graphical example of the decoding algorithm

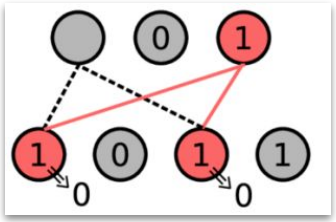
Let us see the full example:

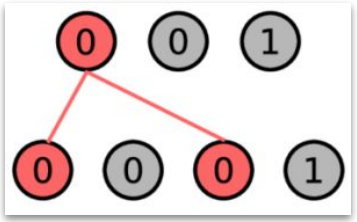






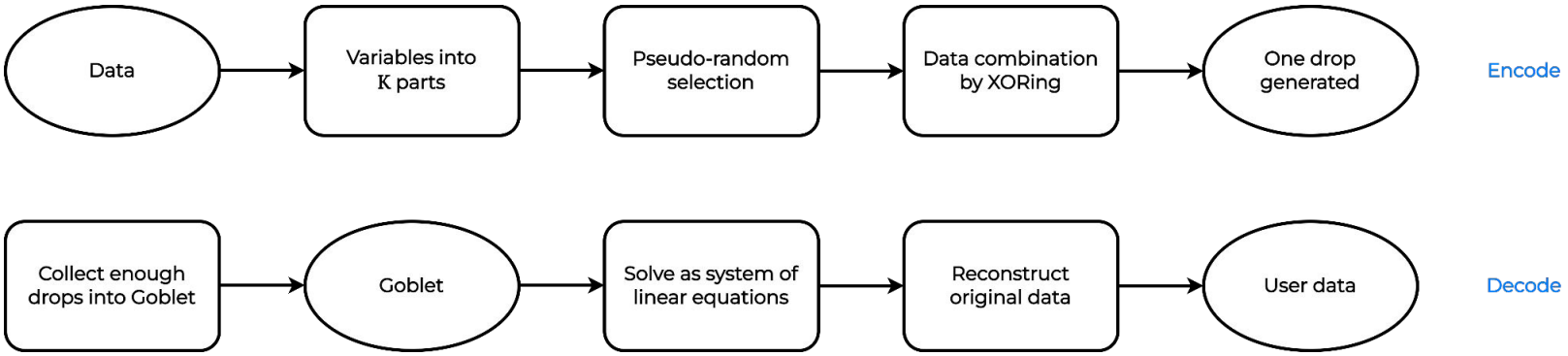






This mechanism leads to the decrease of degrees of all the neighbouring output symbols. Now, we have more output symbols of degree one and the process continue

Encoding and decoding algorithms for fountain codes



Through these two **flowcharts** we can summarize Luby's encoding and decoding algorithms

Why LT codes?

- Understanding it with two examples
- Problem & Solution

“Classic” packets transmission

To fully understand why fountain codes are widely used in practice (i.e. Luby codes), let us see how packet transmission works

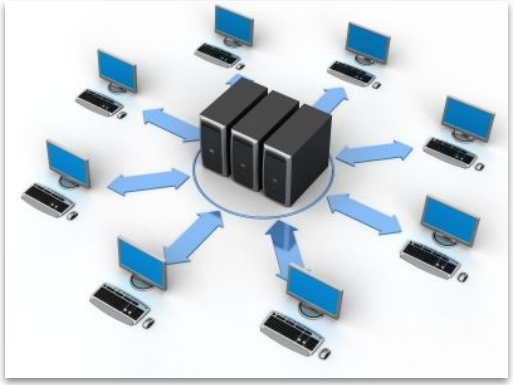
The **traditional file transmission mechanism** is as follows:

1. A file is divided into equal length packets
2. Transmitter sends packets one by one
3. Receiver acknowledges each received packet as well as lost ones
4. Transmitter re-sends packets lost

“Classic” packets transmission: example problem

Now consider a case where a **server** is to **distribute data** to **multiple receivers**, the content can be operating system updates or media files, anything

Using the traditional method, not only will **the server have to keep a duplex communication with the clients**, **it will also have to maintain a list of which client got which packets**, in order to transmit lost ones again



“Classic” packets transmission: example problem

Just to do another example: consider a **NASA** expedition on a **mission** to **Mars** Despite of the very high computing power, the **connection** to **earth** is **very lossy**

Since many of the **packets sent** are **lost**, and the **receiver** cannot present **any feedback** of which packets to be re-transmitted

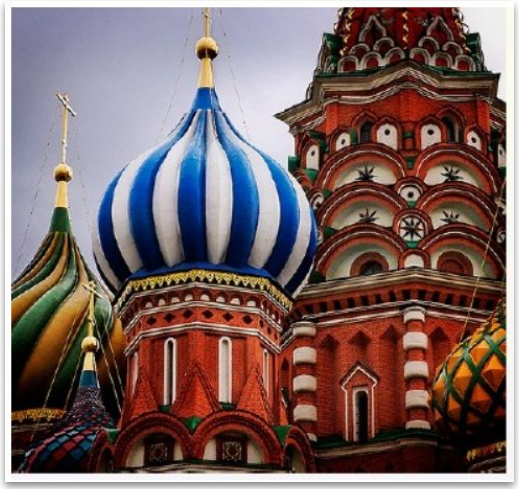
The solution is the concept of digital fountain

In a digital fountain, **it doesn't matter which packets are received and which are lost, it also doesn't matter the order of received packets**

The **server transformers** the data into **unlimited number** of **encoded chunks**, then the **client** can **reassemble** the data **given** enough **number** of **chunks**, regardless of which ones were get and which ones were missed

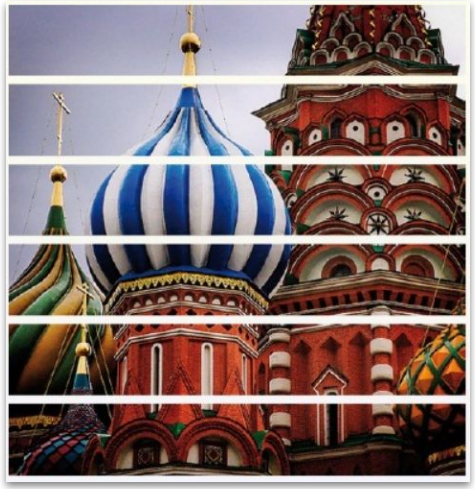
That's why it was called a **fountain**, you **don't care** which **drops** you **get** from the **fountain**, as long as you get enough drops to fill your bottle

Working example



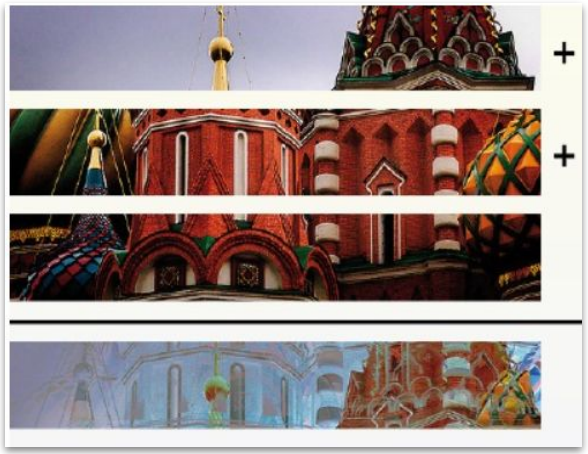
Consider this image as a source of information to be transmitted

Working example: encode



Split into $k = 6$ parts

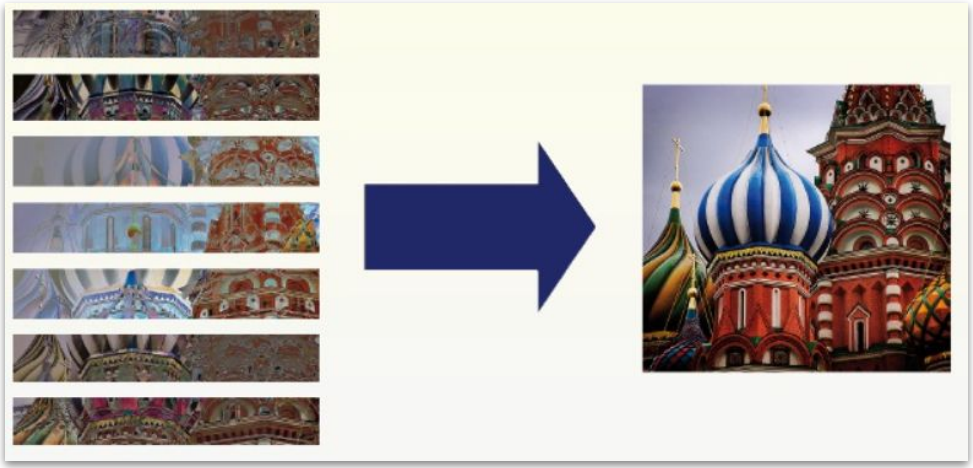
One drop is generated by XORing parts 1, 2 and 3



Another drop is generated by XORing parts 1, 5 and 6

Drop generation can be repeated over and over again, until necessary

Working example: decode



This is the **same** as **solving** an **equation system** with K unknowns (data parts)

Every **drop** is an **equation** of the system

The file encoder sends $\mathbf{B}$ **blocks**, but we need only $K<B$ drops (equations)

Degree distribution of LT codes

- Ideal soliton distribution
- Robust soliton distribution

Degree distribution

When we explained how the algorithm works, we assumed that a degree distribution $\rho(d)$ was provided

This is a discrete *probability distribution* (**probability mass function**) on integers between 1 and k

How the distribution $\rho(d)$ has to be designed?

Ideal soliton distribution

This following distribution **theoretically** minimizes the expected number of **redundant** code words that will be sent before the decoding process can be completed

$$\rho(1) = 1/k$$
$$\text{For all } i = 2, \ldots, k \rightarrow \rho(i) = 1/i(i-1)$$

However the ideal soliton distribution does not work well in practice because any fluctuation around the expected behavior makes it likely that at some step in the decoding process there will be **no available packet of (reduced) degree 1** so decoding will fail

It is useless in practice!

Robust soliton distribution

We introduce the **robust soliton distribution (RSD)**.

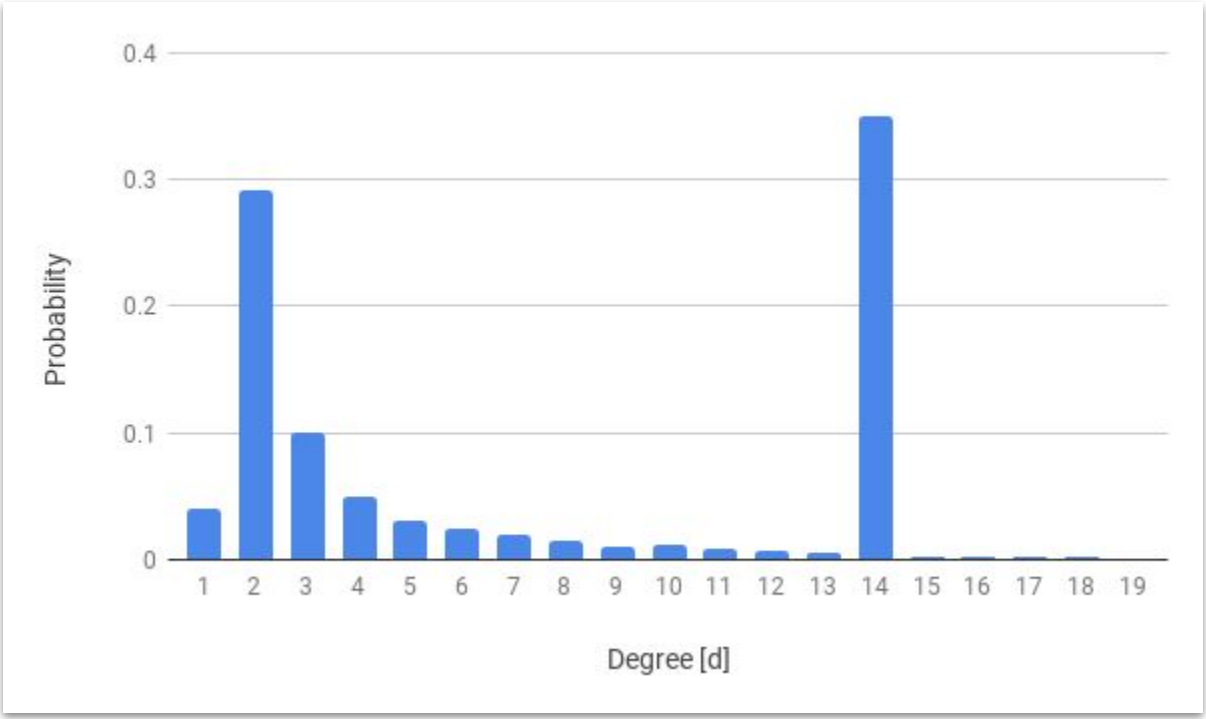
It has two extra parameters, c (a suitable > 0 constant) and δ (allowable failure probability) and it is designed to produce more packets of very small degree (around 1) and fewer packets of degree greater than 1, except for a **spike** of packets at a fairly large quantity chosen to ensure that all original blocks will be included in some packet

$$\boxed{\text{For all } i = 1, \ldots, k \rightarrow \mu(i) = (\rho(i) + \tau(i))/\beta}$$

$$\tau(i) = \begin{cases} R/ik & \text{for } i = 1, \ldots, k/R - 1 \\ R\ln(R/\delta)/k & \text{for } i = k/R \\ 0 & \text{for } i = k/R + 1, \ldots, k \end{cases} \qquad \beta = \sum_{i=1}^{k} \rho(i) + \tau(i) \qquad R = c \ln\left(\tfrac{k}{\delta}\right)\sqrt{k}$$

Robust soliton distribution

E.g. $k = 1000,\ \delta = 0.01,\ c = 0.02$



Luby’s main theorem

Luby’s main theorem proves that there exists a value of $\mathbf{c}$ such that, given $\mathbf{K}$ received packets, the decoding algorithm will recover the $\mathbf{k}$ source packets with probability greater than $\mathbf{1 - \delta}$

Formally:

$$\exists\, c > 0 : \Pr[\text{decoding succeeds with K encoding symbols}] \geq 1 - \delta$$
$$\text{where } \mathrm{K} = \mathrm{k} + \sqrt{\mathrm{k}}\, \ln(\mathrm{k}/\delta)$$

We’ll show later some trials showing how **performances** changes changing the discussed **distribution parameters**

The importance of degree distribution

The distribution used for generating the output symbols is the **heart of LT-codes**

If the decoding algorithm is successful depends solely on the output degree distribution

Our implementation of LT codes

Symbol class

```
class Symbol:
    __slots__ = ["index", "degree", "data", "neighbours"]


    def __init__(self, index, degree, data):
        self.index = index
        self.degree = degree
        self.data = data
```

This attribute indicates the **index** of the **block** that has been **produced**

This attribute contains the **indexes** that were **used** for **symbol encoding**

This attribute contains the **result** of the **XOR operation**

This attribute **indicates** the **degree chosen** by the **encode function** that is needed to produce the **encoding symbol**

We have represented **the concept of the symbol transmitted through a Symbol class** which maintains the attributes necessary for the encoding of each symbol

Encode

```python
def encode(blocks, c=None, delta=DELTA, debug=True):
    """..."""
    blocks_n = len(blocks)
    p_dist = robust_distribution(blocks_n, c, delta)

    i = 0
    while True:
        # Generate random index associated to random degree, seeded with the symbol id
        d = get_degree_from(p_dist)
        # Get the random selection
        selection_indexes, d = generate_indexes(i, d, blocks_n)
        # Xor each selected array within each other
        symbol_data = blocks[selection_indexes[0]]
        for n in range(1, d):
            symbol_data = np.bitwise_xor(symbol_data, blocks[selection_indexes[n]])
        # Create symbol, then log the process
        symbol = Symbol(index=i, degree=d, data=symbol_data)
        if debug:
            print("[ENCODER] Released symbol", i)
        i += 1

        yield symbol
```

The parameter **blocks** is an **array** of **bytes** containing each block of the **source data**

The parameters c, and delta are used to **generate** the **Robust Soliton Distribution**

Encode

```
def encode(blocks, c=None, delta=DELTA, debug=True):
    """..."""
    blocks_n = len(blocks)
    p_dist = robust_distribution(blocks_n, c, delta)

    i = 0
    while True:
        # Generate random index associated to random degree, seeded with the symbol id
        d = get_degree_from(p_dist)
        # Get the random selection
        selection_indexes, d = generate_indexes(i, d, blocks_n)
        # Xor each selected array within each other
        symbol_data = blocks[selection_indexes[0]]
        for n in range(1, d):
            symbol_data = np.bitwise_xor(symbol_data, blocks[selection_indexes[n]])
        # Create symbol, then log the process
        symbol = Symbol(index=i, degree=d, data=symbol_data)
        if debug:
            print("[ENCODER] Released symbol", i)
        i += 1

        yield symbol
```

blocks_n contains the **number** of **blocks** into which the **source** has been **divided**

The encoder needs to generate a **Robust Soliton Distribution**

The while True construct represents the **concept** that the **encoder produces** a potentially **infinite number** of **symbols**

Encode

```
def encode(blocks, c=None, delta=DELTA, debug=True):
    """..."""
    blocks_n = len(blocks)
    p_dist = robust_distribution(blocks_n, c, delta)

    i = 0
    while True:
        # Generate random index associated to random degree, seeded with the symbol id
        d = get_degree_from(p_dist)
        # Get the random selection
        selection_indexes, d = generate_indexes(i, d, blocks_n)
        # Xor each selected array within each other
        symbol_data = blocks[selection_indexes[0]]
        for n in range(1, d):
            symbol_data = np.bitwise_xor(symbol_data, blocks[selection_indexes[n]])
        # Create symbol, then log the process
        symbol = Symbol(index=i, degree=d, data=symbol_data)
        if debug:
            print("[ENCODER] Released symbol", i)
        i += 1

        yield symbol
```

d is the **degree** generated by the **degree distribution** p_dist

The **encoder generates** the **indexes** of the blocks to be **added mod 2**

The generate_indexes function does it **uniformly**

Encode

```
def encode(blocks, c=None, delta=DELTA, debug=True):
    """..."""
    blocks_n = len(blocks)
    p_dist = robust_distribution(blocks_n, c, delta)

    i = 0
    while True:
        # Generate random index associated to random degree, seeded with the symbol id
        d = get_degree_from(p_dist)
        # Get the random selection
        selection_indexes, d = generate_indexes(i, d, blocks_n)
        # Xor each selected array within each other
        symbol_data = blocks[selection_indexes[0]]
        for n in range(1, d):
            symbol_data = np.bitwise_xor(symbol_data, blocks[selection_indexes[n]])
        # Create symbol, then log the process
        symbol = Symbol(index=i, degree=d, data=symbol_data)
        if debug:
            print("[ENCODER] Released symbol", i)
        i += 1

        yield symbol
```

Note that the list of indexes of the blocks used for the XOR operation is not passed to the symbol object

The **encoder XORs** the **blocks** whose indexes have been **chosen uniformly**

The encode function has the structure of a **Python generator**.
At each iteration a **new symbol** is **generated**

Decode

How does the encoder know the indexes of the blocks that have been added (mod 2)?

```
def generate_indexes(symbol_index, degree, blocks_quantity):

    random.seed(symbol_index*seed_r)
    indexes = random.sample(range(blocks_quantity), degree)
```

The answer lies in how the random function works: if the **same seed** is passed, the function generates the same sequence of pseudo-random numbers

The **seed** is the index of the block that was produced

Decode

- The decode takes in input the list of symbols
- It returns how many blocks it has decoded (solved_block_count)

If the decoder fails to decode, it means that it has to wait for the encoder to produce more symbols

```python
def decode(symbols, blocks, solved_blocks_count, debug=True):
    blocks_n = len(blocks)

    while True:
        # num of block solved in this iteration
        iteration_solved_count = 0
        # Search for solvable symbols
        for i, symbol in enumerate(symbols):
            # Check the current degree. If it's 1 then we can recover data
            if symbol.degree == 1:
                iteration_solved_count += 1
                block_index = next(iter(symbol.neighbours))
                symbols.pop(i)
                if blocks[block_index] is not None: # This symbol is redundant
                    continue
                blocks[block_index] = symbol.data
                solved_blocks_count += 1
                if debug:
                    print("[DECODER] Solved block", block_index)
                    print("          Currently solved", solved_blocks_count, "/", blocks_n, "blocks.")
                # Reduce the degrees of other symbols that contains the solved block as neighbor
                reduce_neighbors(block_index, blocks, symbols)
        if iteration_solved_count == 0:
            break

    return np.asarray(blocks, dtype=object), solved_blocks_count, symbols
```

iteration_solved_count indicates **how many blocks are decoded in the current iteration**

Decode

The decoder iterates through all the symbols. If it finds one of **degree 1** returns it in output and increases iteration_solved_count

The decoder takes the index of its neighbour (itself)

```python
def decode(symbols, blocks, solved_blocks_count, debug=True):
    blocks_n = len(blocks)

    while True:
        # num of block solved in this iteration
        iteration_solved_count = 0
        # Search for solvable symbols
        for i, symbol in enumerate(symbols):
            # Check the current degree. If it's 1 then we can recover data
            if symbol.degree == 1:
                iteration_solved_count += 1
                block_index = next(iter(symbol.neighbours))
                symbols.pop(i)
                if blocks[block_index] is not None: # This symbol is redundant
                    continue
                blocks[block_index] = symbol.data
                solved_blocks_count += 1
                if debug:
                    print("[DECODER] Solved block", block_index)
                    print("          Currently solved", solved_blocks_count, "/", blocks_n, "blocks.")
                # Reduce the degrees of other symbols that contains the solved block as neighbor
                reduce_neighbors(block_index, blocks, symbols)
        if iteration_solved_count == 0:
            break

    return np.asarray(blocks, dtype=object), solved_blocks_count, symbols
```

Decode

At the beginning, all the blocks are initialized with **None** because nothing has been decoded yet

Therefore, there will be **None** for the blocks not decoded yet and the **symbol data** for those that have been decoded

```
def decode(symbols, blocks, solved_blocks_count, debug=True):
    blocks_n = len(blocks)

    while True:
        # num of block solved in this iteration
        iteration_solved_count = 0
        # Search for solvable symbols
        for i, symbol in enumerate(symbols):
            # Check the current degree. If it's 1 then we can recover data
            if symbol.degree == 1:
                iteration_solved_count += 1
                block_index = next(iter(symbol.neighbours))
                symbols.pop(i)
                if blocks[block_index] is not None: # This symbol is redundant
                    continue
                blocks[block_index] = symbol.data
                solved_blocks_count += 1
                if debug:
                    print("[DECODER] Solved block", block_index)
                    print("          Currently solved", solved_blocks_count, "/", blocks_n, "blocks.")
                # Reduce the degrees of other symbols that contains the solved block as neighbor
                reduce_neighbors(block_index, blocks, symbols)
        if iteration_solved_count == 0:
            break

    return np.asarray(blocks, dtype=object), solved_blocks_count, symbols
```

Decode

Reduces the degrees of the other symbols that contain the solved block as neighbour

If the decoder hasn't solved any blocks in the **current iteration** it breaks, otherwise it continues

It returns how many blocks have been currently solved and the current status of the blocks

```python
def decode(symbols, blocks, solved_blocks_count, debug=True):
    blocks_n = len(blocks)

    while True:
        # num of block solved in this iteration
        iteration_solved_count = 0
        # Search for solvable symbols
        for i, symbol in enumerate(symbols):
            # Check the current degree. If it's 1 then we can recover data
            if symbol.degree == 1:
                iteration_solved_count += 1
                block_index = next(iter(symbol.neighbours))
                symbols.pop(i)
                if blocks[block_index] is not None: # This symbol is redundant
                    continue
                blocks[block_index] = symbol.data
                solved_blocks_count += 1
                if debug:
                    print("[DECODER] Solved block", block_index)
                    print("          Currently solved", solved_blocks_count, "/", blocks_n, "blocks.")
                # Reduce the degrees of other symbols that contains the solved block as neighbor
                reduce_neighbors(block_index, blocks, symbols)
        if iteration_solved_count == 0:
            break

    return np.asarray(blocks, dtype=object), solved_blocks_count, symbols
```

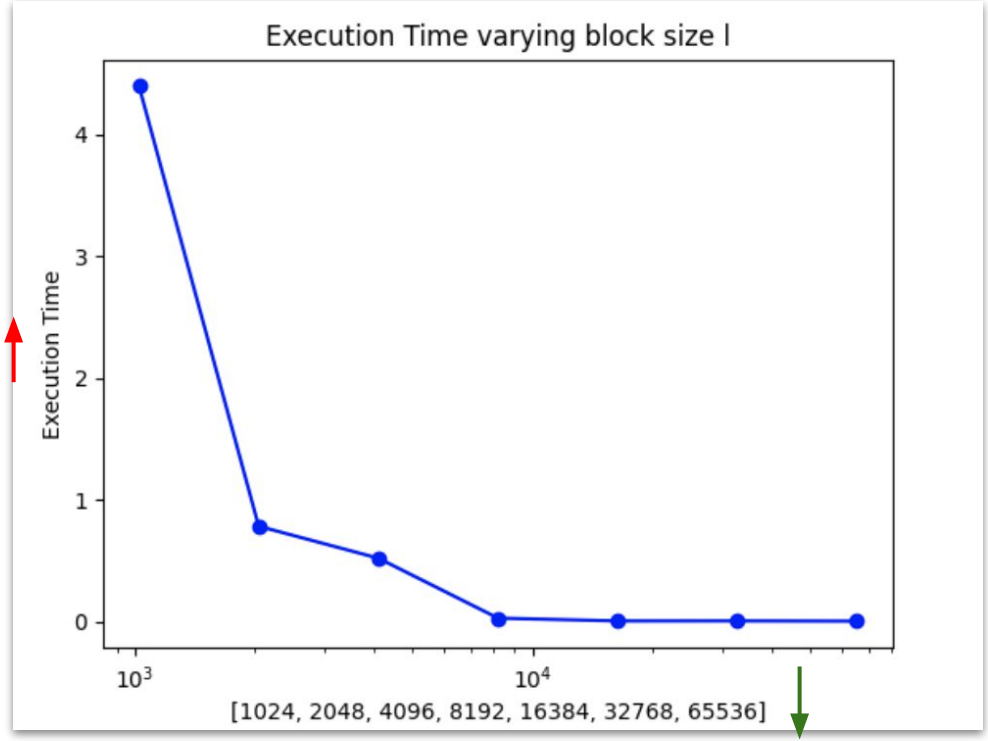
Slowed down simulation with packet loss

Some benchmark

- Experimenting how block size influences execution time
- Experimenting how distributions parameters influences code success rate
- Experimenting how the codes perform introducing packets loss

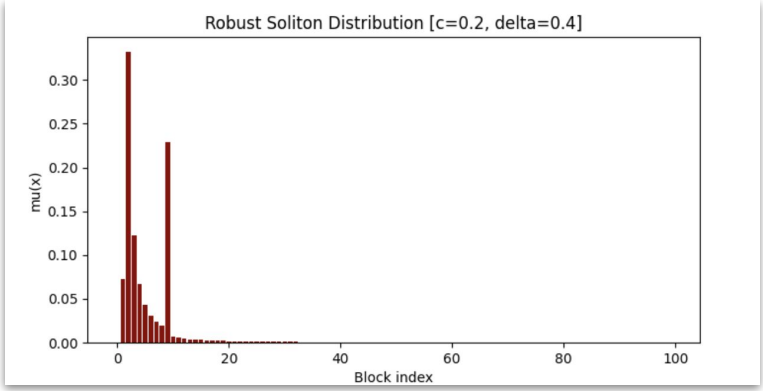
How block size influences execution time

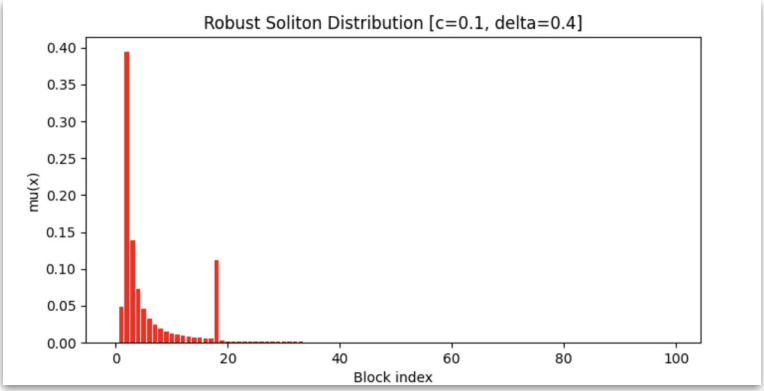
```
Input size: 488.28 KB

Block size l: 1024
Blocks: 489
Simulation in progress...
Execution time: 4.3998 seconds

Block size l: 2048
Blocks: 245
Simulation in progress...
Execution time: 0.7833 seconds

...

Block size l: 65536
Blocks: 8
Simulation in progress...
Execution time: 0.0034 seconds
```

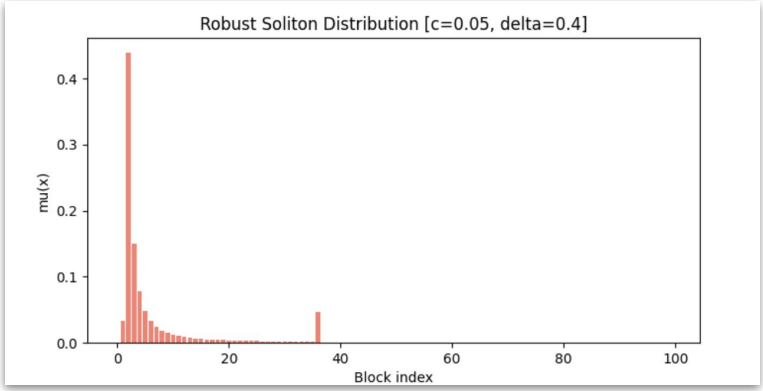


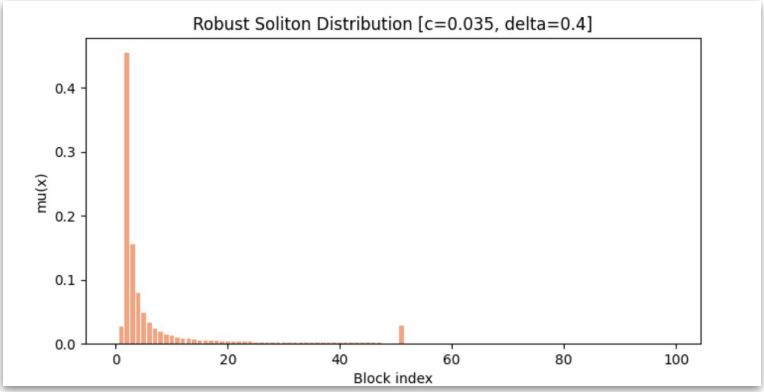
How distributions parameters influences code success rate

$$k = 100 \quad \delta = 0.4$$









How distributions parameters influences code success rate

We want to show how RSD **parameters** influence the performance of LT codes

We run a simulation for N_TRIALS times (1000)

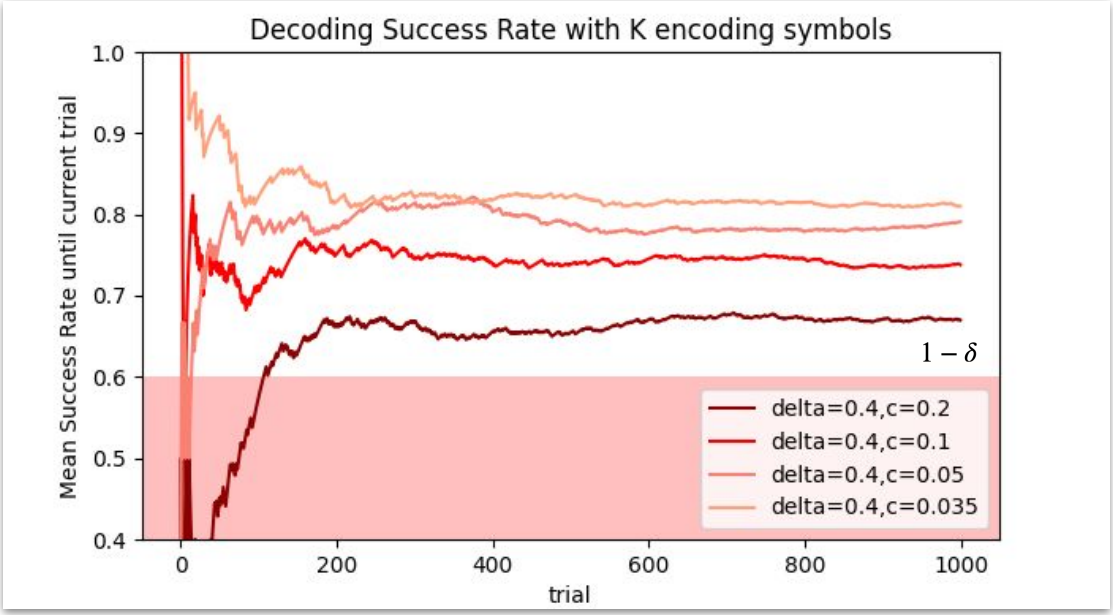
The simulation consists of the **encoder**, producing $\mathbf{K}$ symbols and **decoder** decoding them. If these encoding symbols were enough for the decoder to decode the original blocks we increment a **counter**

At the end of the trials we divide the counter for the number of trials, obtaining the **average** (decoding) **success rate**

In the following charts we show the evolution, through the trials, of the average success rate. Note that is important to look at the convergence values (where the arithmetic average coincides with the statistic mean)

How distributions parameters influences code success rate

$$k = 100$$



$$\exists\, c > 0 : \Pr[\text{decoding succeeds with K encoding symbols}] \geq 1 - \delta$$
$$\text{where } \mathrm{K} = \mathrm{k} + \sqrt{\mathrm{k}}\,\ln(\mathrm{k}/\delta)$$

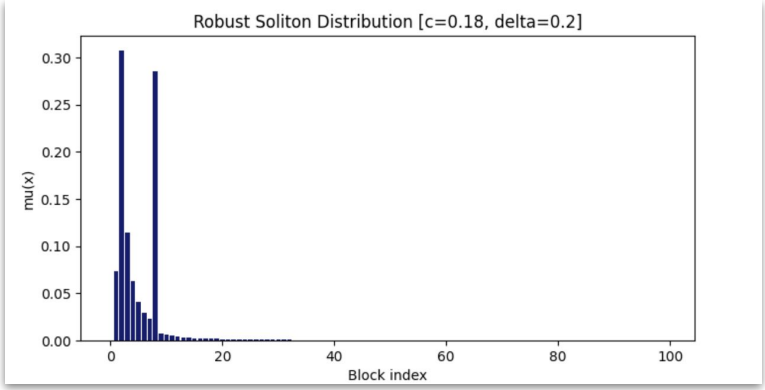
Given a fixed value for the **allowable failure probability** δ, we show the evolution of the **average success rate using K encoding symbols** (iteration by iteration)

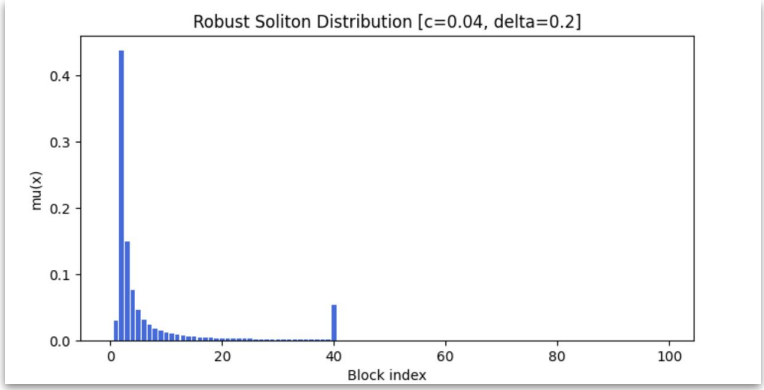
We can see that all the specified values of ***c*** verify the **Luby's main theorem**

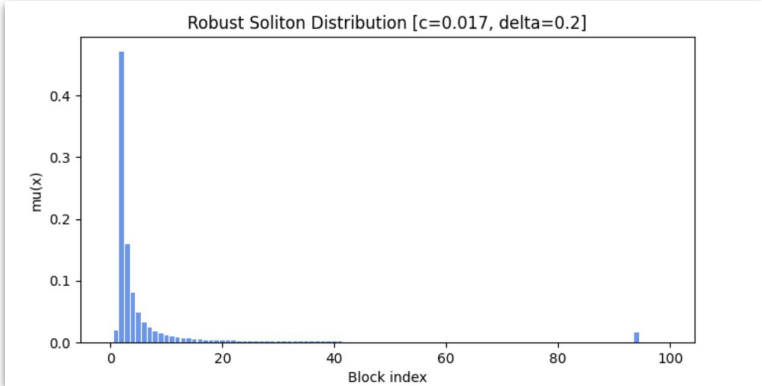
D.J.C. MacKay. Information theory, inference, and learning algorithms. Cambridge Univ Pr, 2003.

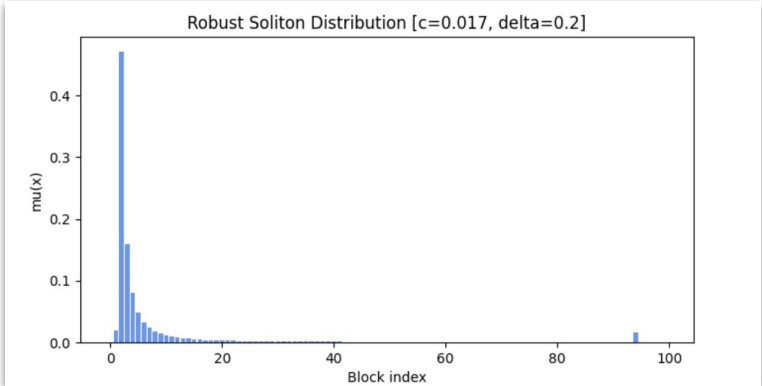
How distributions parameters influences code success rate

$$k = 100 \quad \delta = 0.2$$



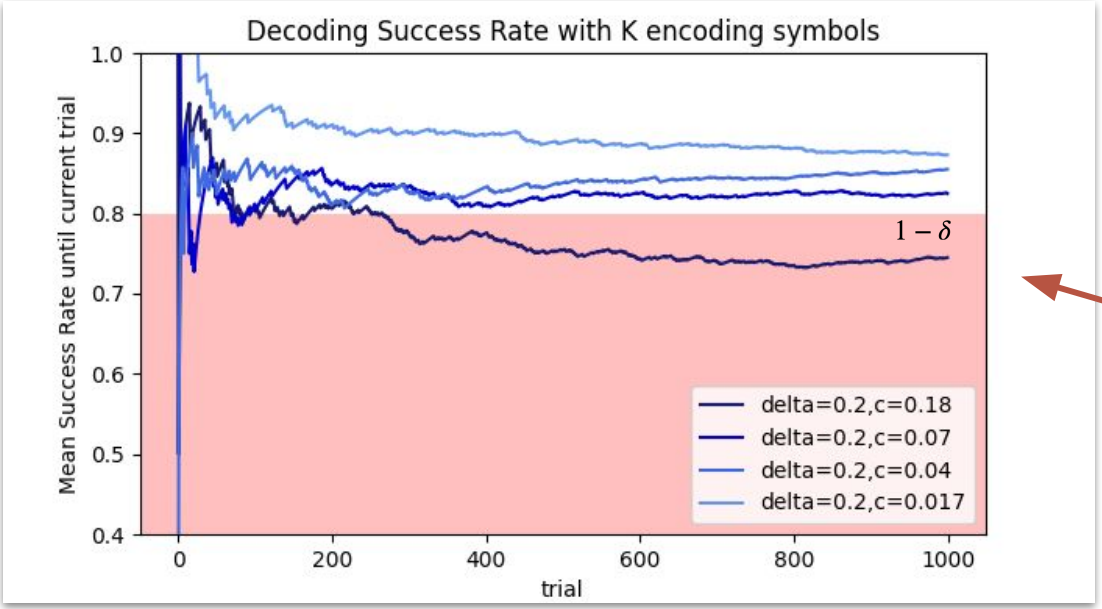






How distributions parameters influences code success rate

$$k = 100$$



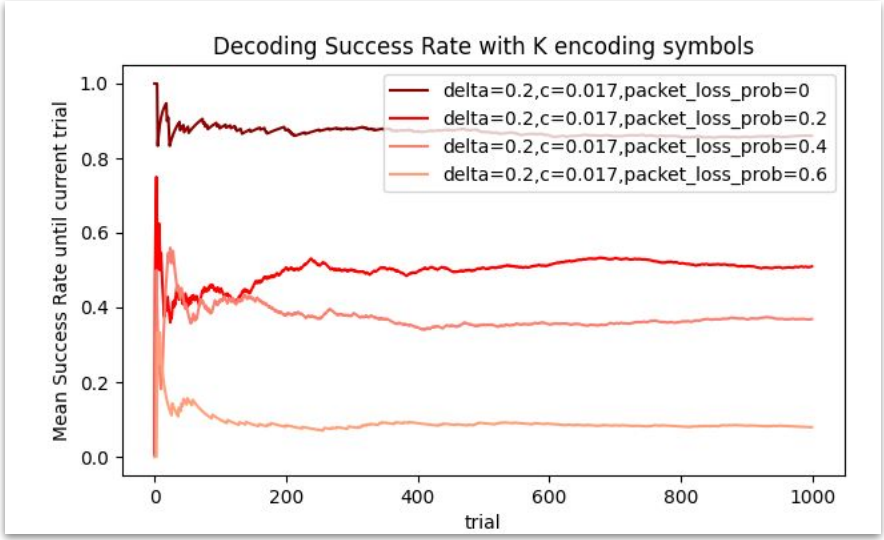
For this value of c the average success rate does not respect the **Luby's main theorem**

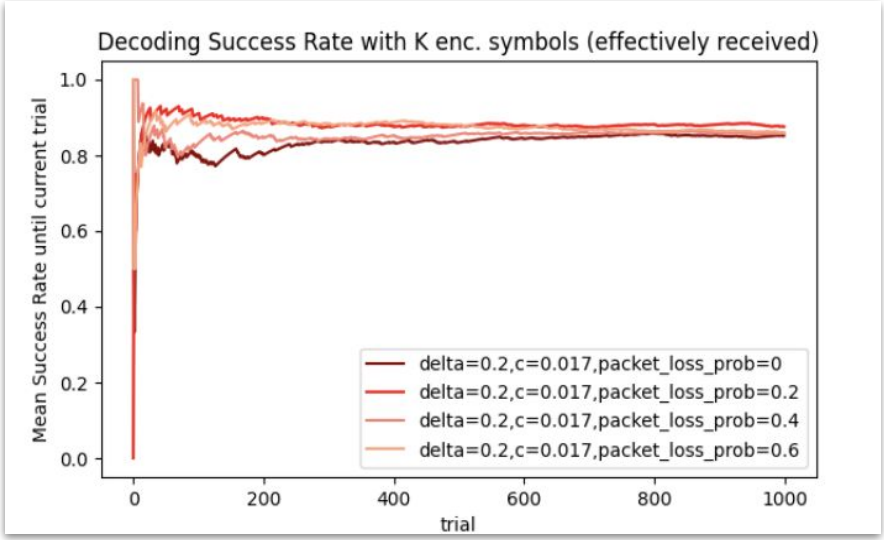
$$\exists\, c > 0 : \Pr[\text{decoding succeeds with K encoding symbols}] \geq 1 - \delta$$
$$\text{where } \mathrm{K} = \mathrm{k} + \sqrt{\mathrm{k}}\,\ln(\mathrm{k}/\delta)$$

D.J.C. MacKay. Information theory, inference, and learning algorithms. Cambridge Univ Pr, 2003.

How the codes perform introducing packets loss

$$k = 100$$



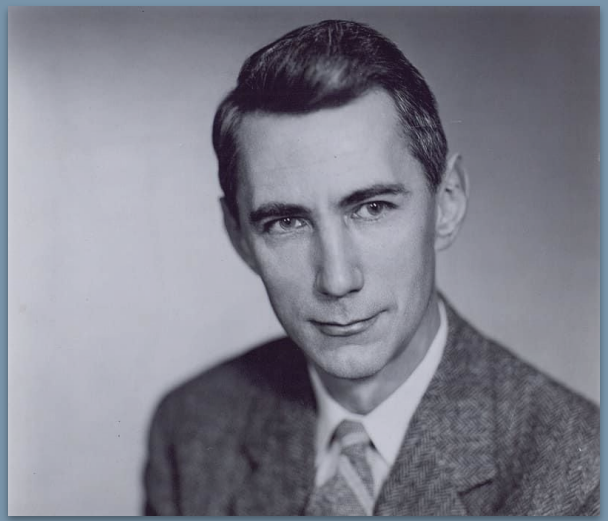


Given a fixed parameter set and number of released encoding symbols the chart shows how the **mean success rate** evolves. We can see that a greater probability of packet loss causes a smaller mean success rate.

References

- **Fountain Coded Wireless Transmission Model**
  - Indian Journal of Science and Technology, Vol 9(14), DOI: 10.17485/ijst/2016/v9i14/86604, April 2016
- **LT Erasure Codes**
  - A paper dating from 2012 that references to chapter 50 of the book by MacKay
  - D.J.C. MacKay. Information theory, inference, and learning algorithms. Cambridge Univ Pr, 2003.
- **LT Codes**
  - M. Luby, "LT codes," The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings., 2002, pp. 271-280, doi: 10.1109/SFCS.2002.1181950.

Thanks for the attention and for this course



“We know the past, but cannot control it.
We control the future, but cannot know it”
- Claude Shannon -