TECHNISCHE UNIVERSITÄT DRESDEN

ComNets
Deutsche Telekom Chair of Communication Networks

Deutsche Telekom Chair of Communication Networks
Technische Universität Dresden

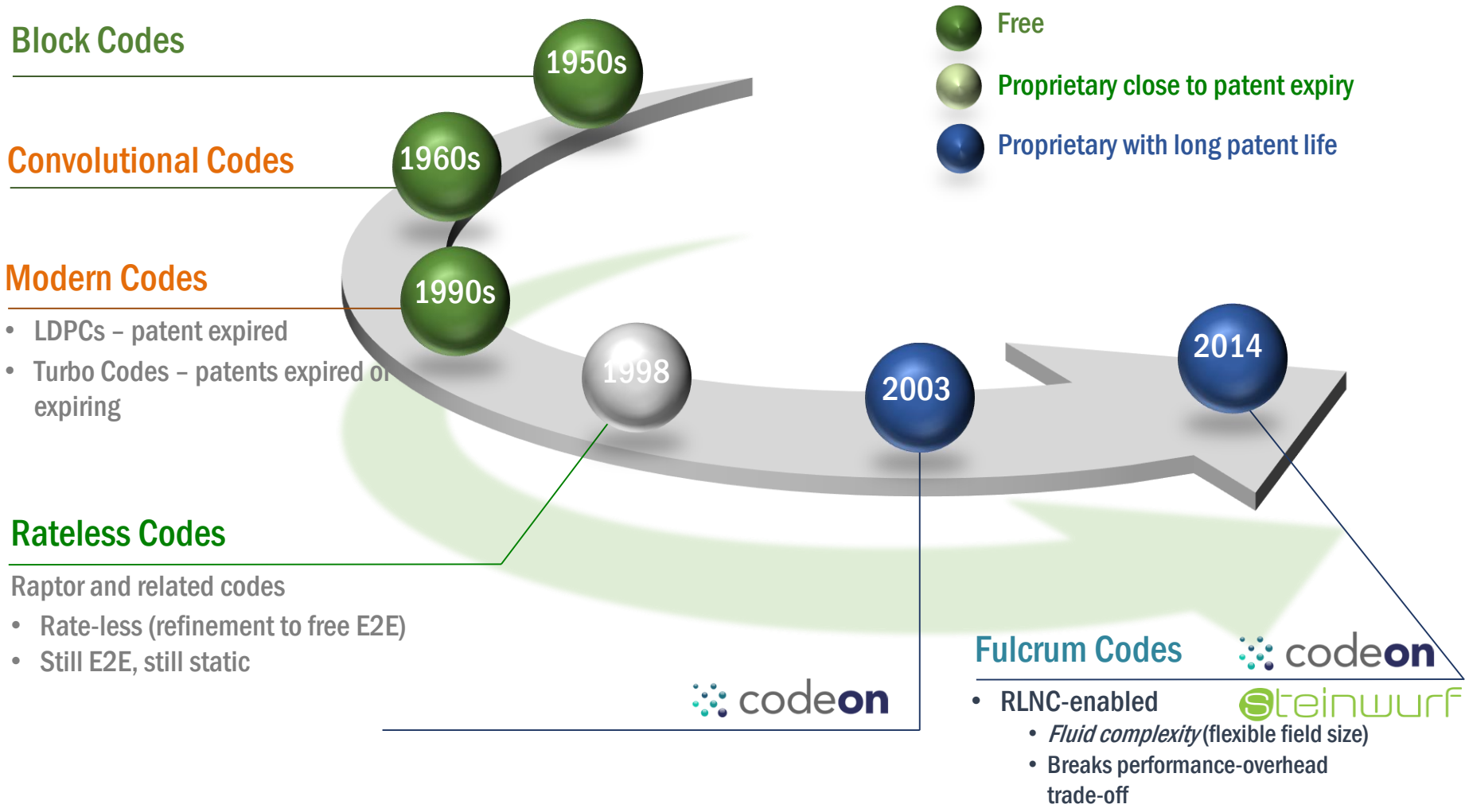
Practical Implementations of Network Coding

Frank Fitzek // Summer Semester 2018

Random Linear Network Coding

Coding History

Evolution of Coding



The Technology: RLNC

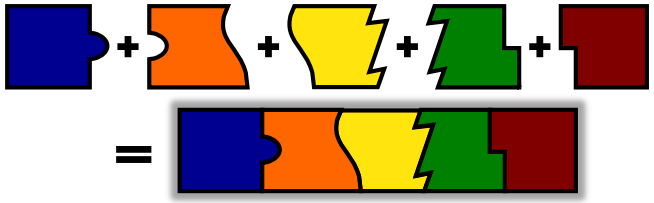
At the heart of many communications problems is a collectors' problem.

Traditional Approach

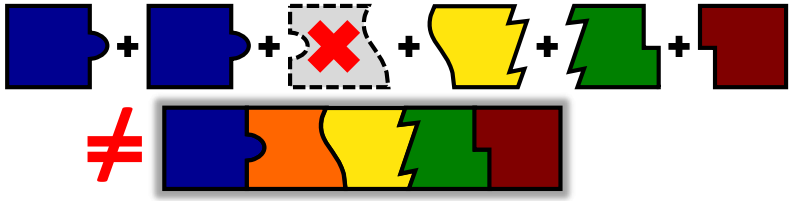
- Data broken into pieces

- k-piece data set → k pieces

- All pieces needed

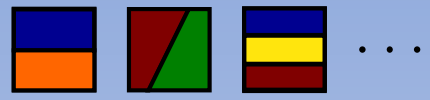


- Only these pieces will do

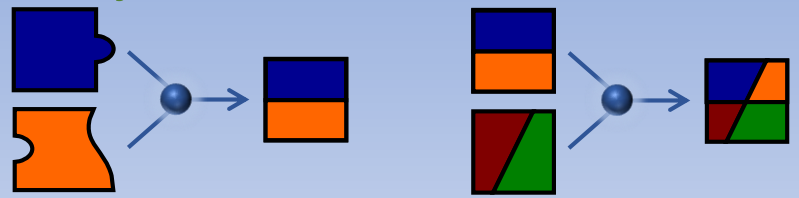


RLNC

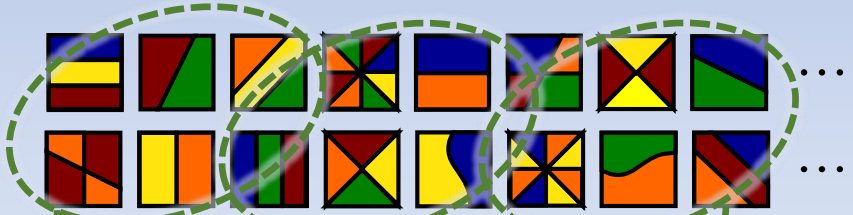
- Mixtures created from pieces



- Any node can create mixtures



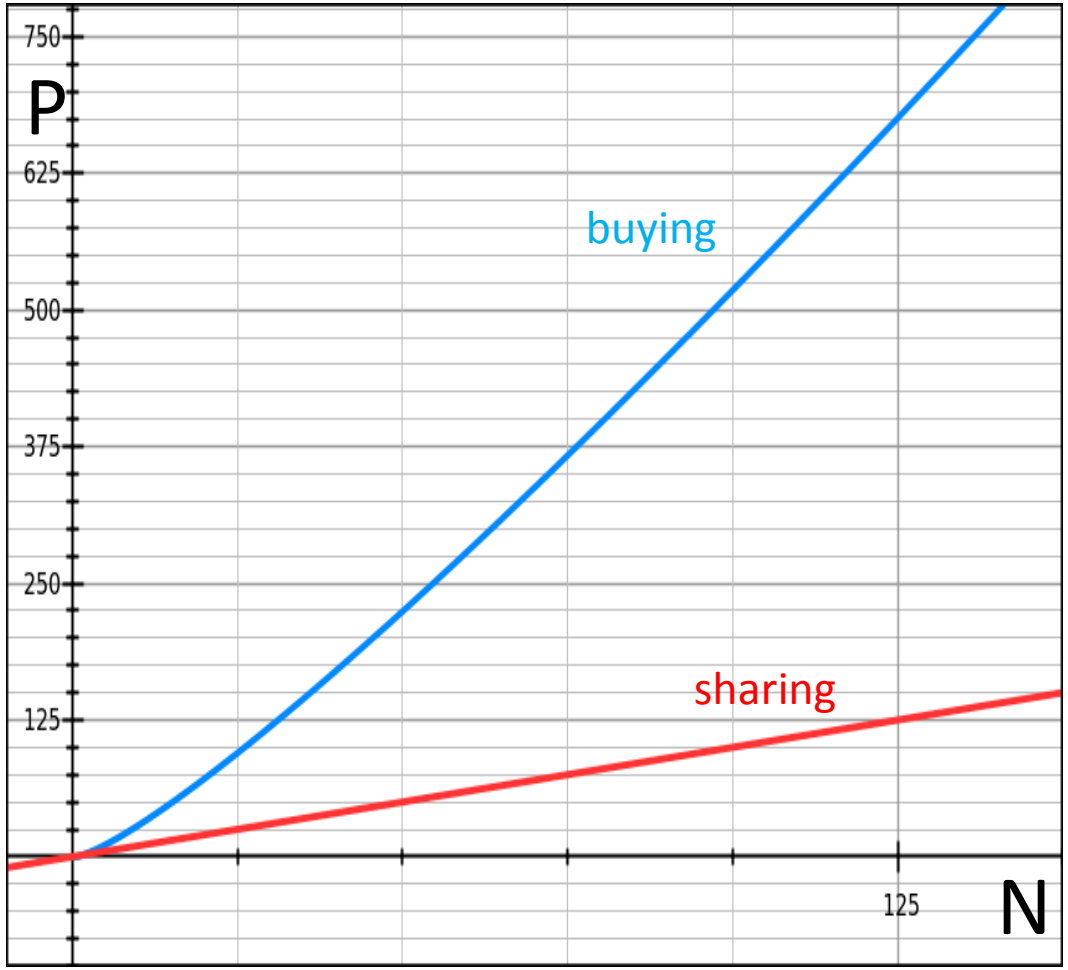
- Many mixtures possible



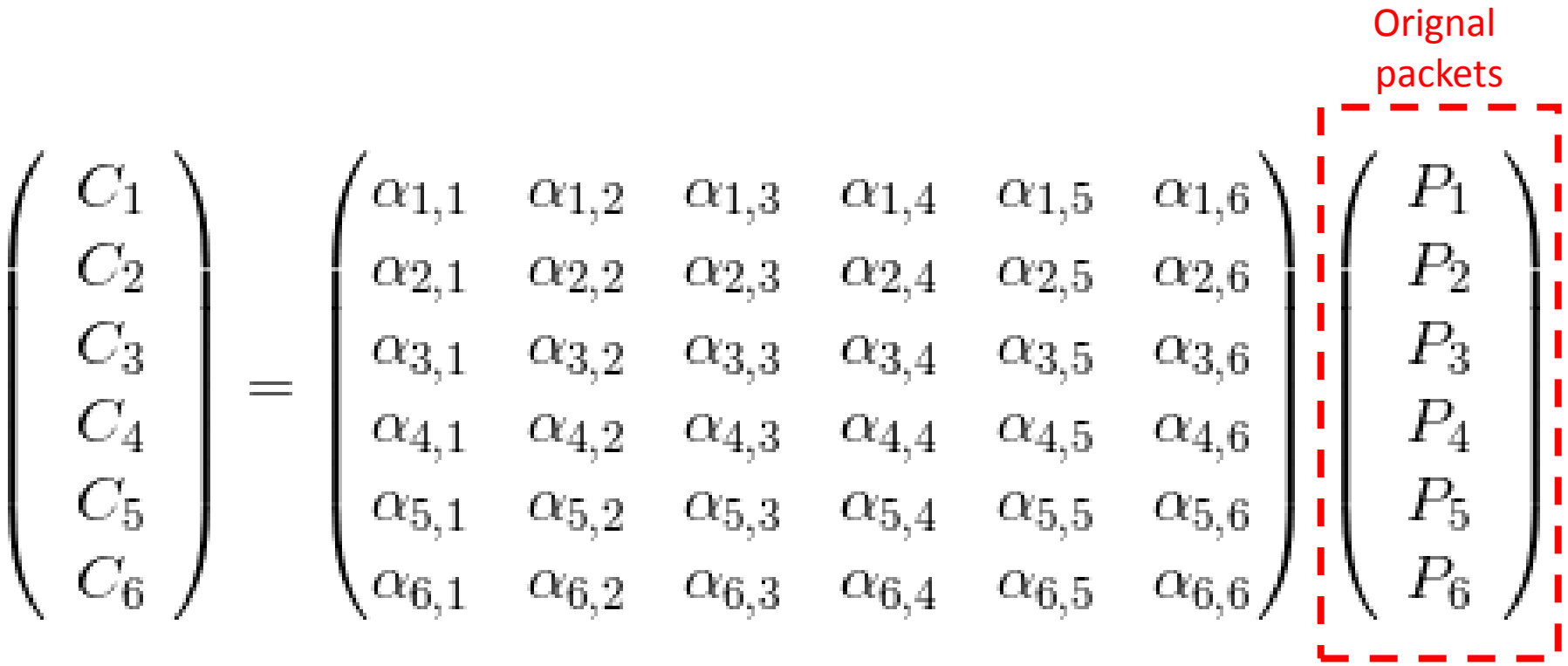
- Any k mixtures will do

Coupon Collector's Problem

$$P = N * (\ln (N) + 0.577)$$



Random Linear Network Coding



Gaussian elimination *n* x *n matrix* requires $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

coding coefficients

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Gaussian elimination *n* x *n matrix* requires $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

coded packets

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & \alpha_{3,5} & \alpha_{3,6} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

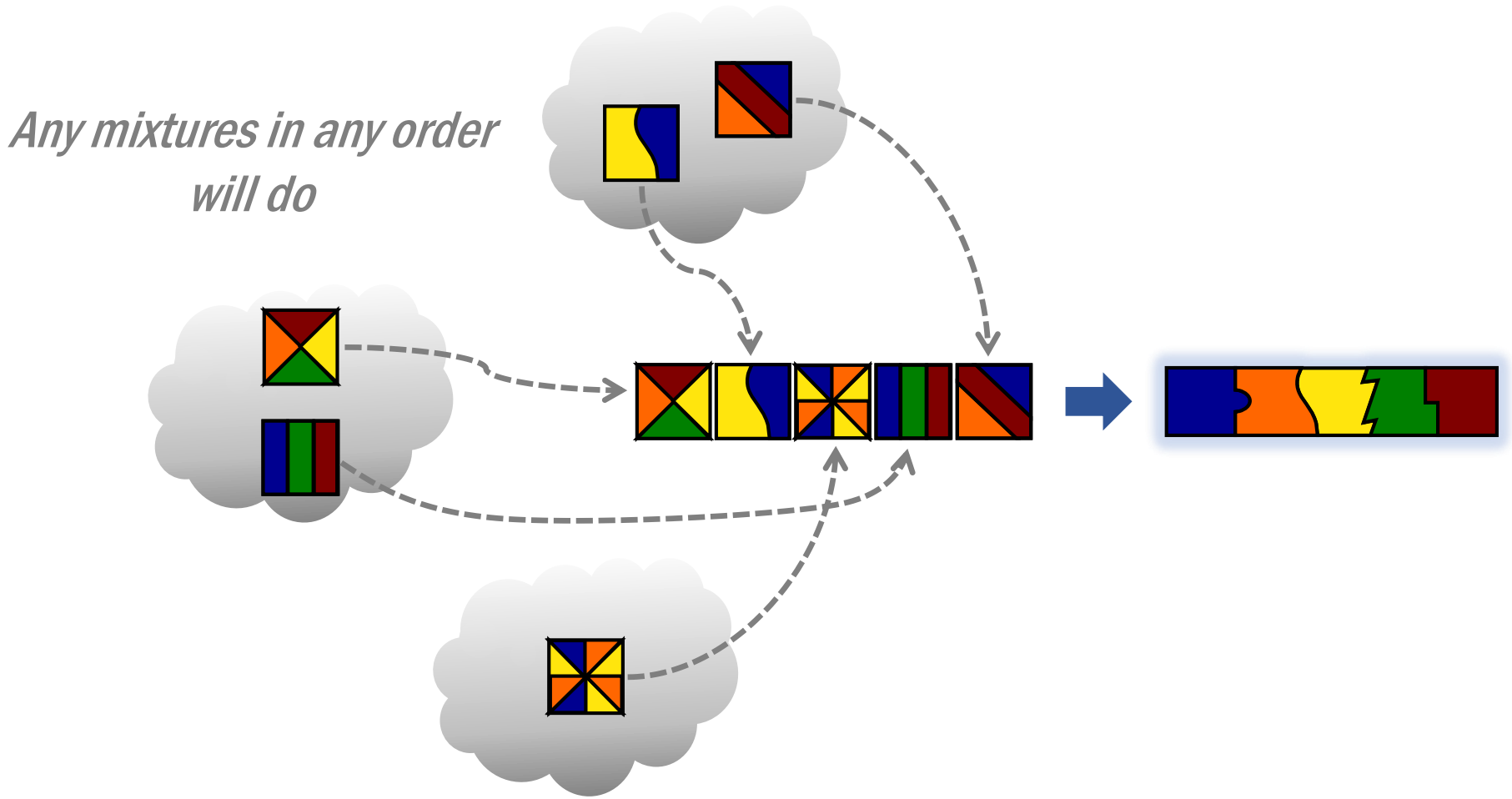
Gaussian elimination *n* x *n matrix* requires $An^3 + Bn^2 + Cn$ operations.

Random Linear Network Coding

$$\begin{pmatrix} C_1 \\ \vdots \\ C_G \\ C_{G+1} \\ \vdots \\ C_K \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \cdots & \alpha_{1,G} \\ \vdots & \ddots & \vdots \\ \alpha_{G,1} & \cdots & \alpha_{G,G} \\ \alpha_{G+1,1} & \cdots & \alpha_{G+1,G} \\ \vdots & \ddots & \vdots \\ \alpha_{K,1} & \cdots & \alpha_{K,G} \end{pmatrix} \begin{pmatrix} P_1 \\ \vdots \\ P_G \end{pmatrix}$$

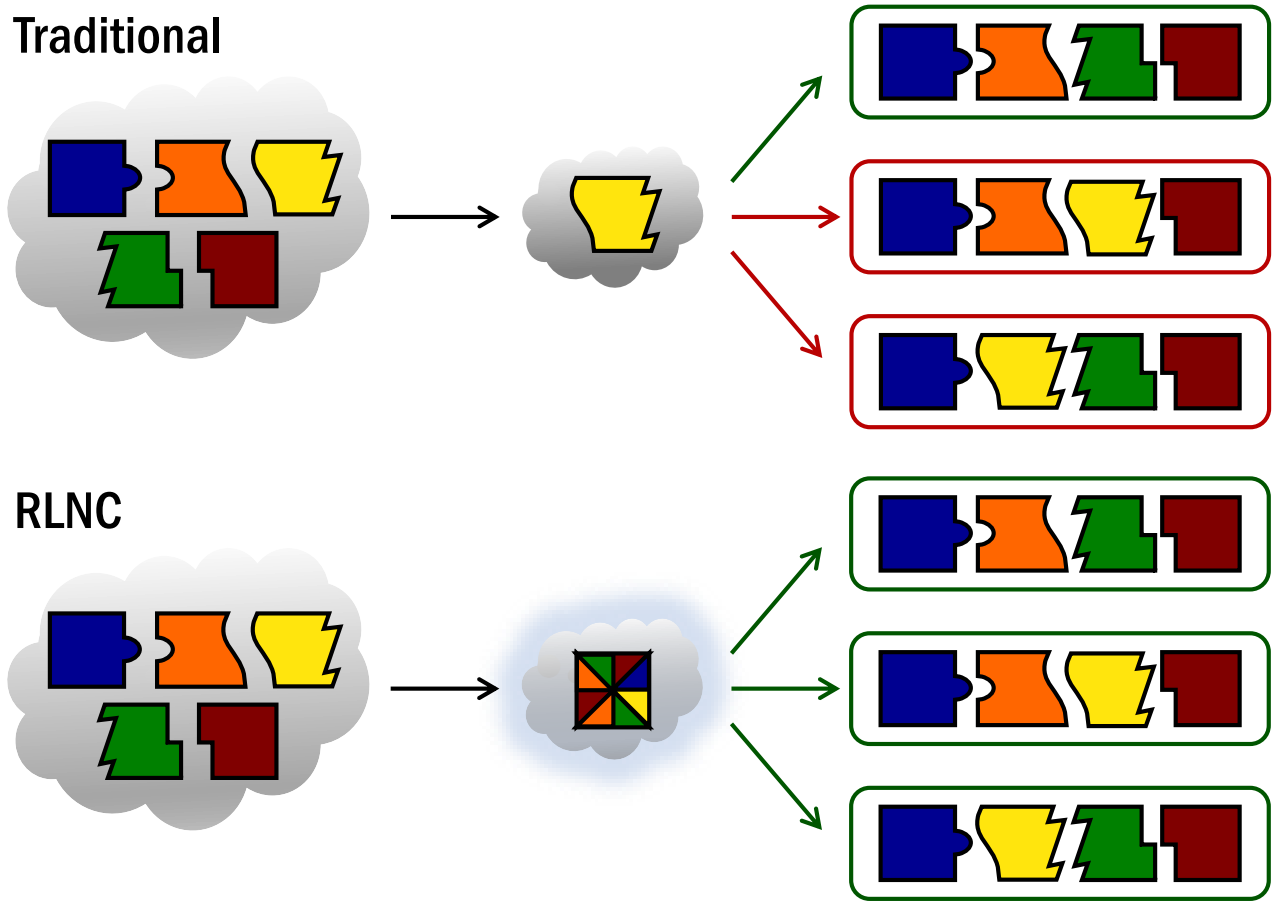
Rateless code: can output any number of coded packets.
(such as Fountain codes, but better than RS)

Multipath – Multicloud



- RLNC Enables “Stateless Communications”

Coded Edge Caching

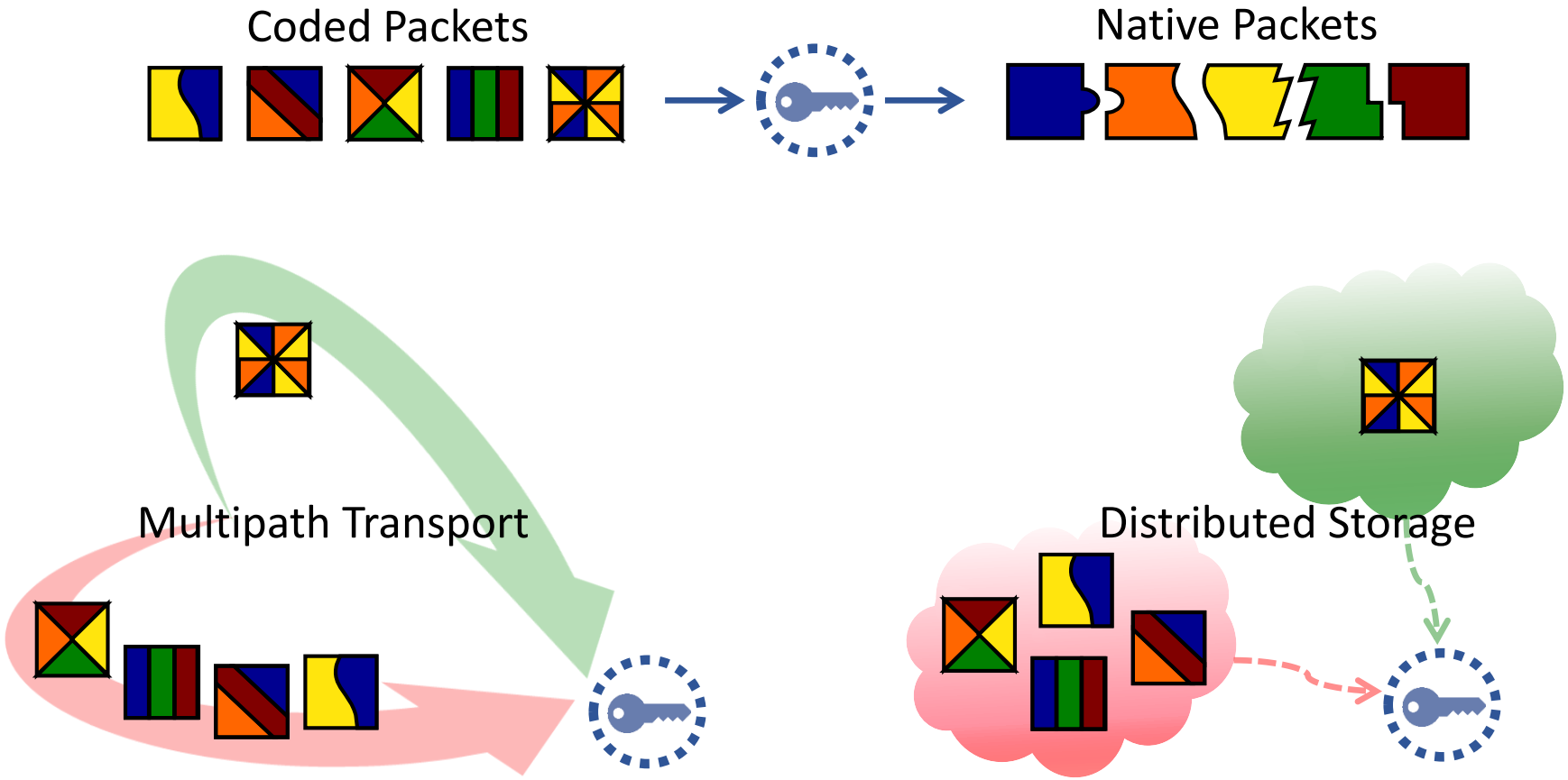


- One small cache can't satisfy everyone
- Only the device missing the piece in the edge cache can reconstruct

- Mixtures enable all nodes to reconstruct
- An example of RLNC's enhancement of non-coded systems

- A Little RLNC Can Go a Long Way

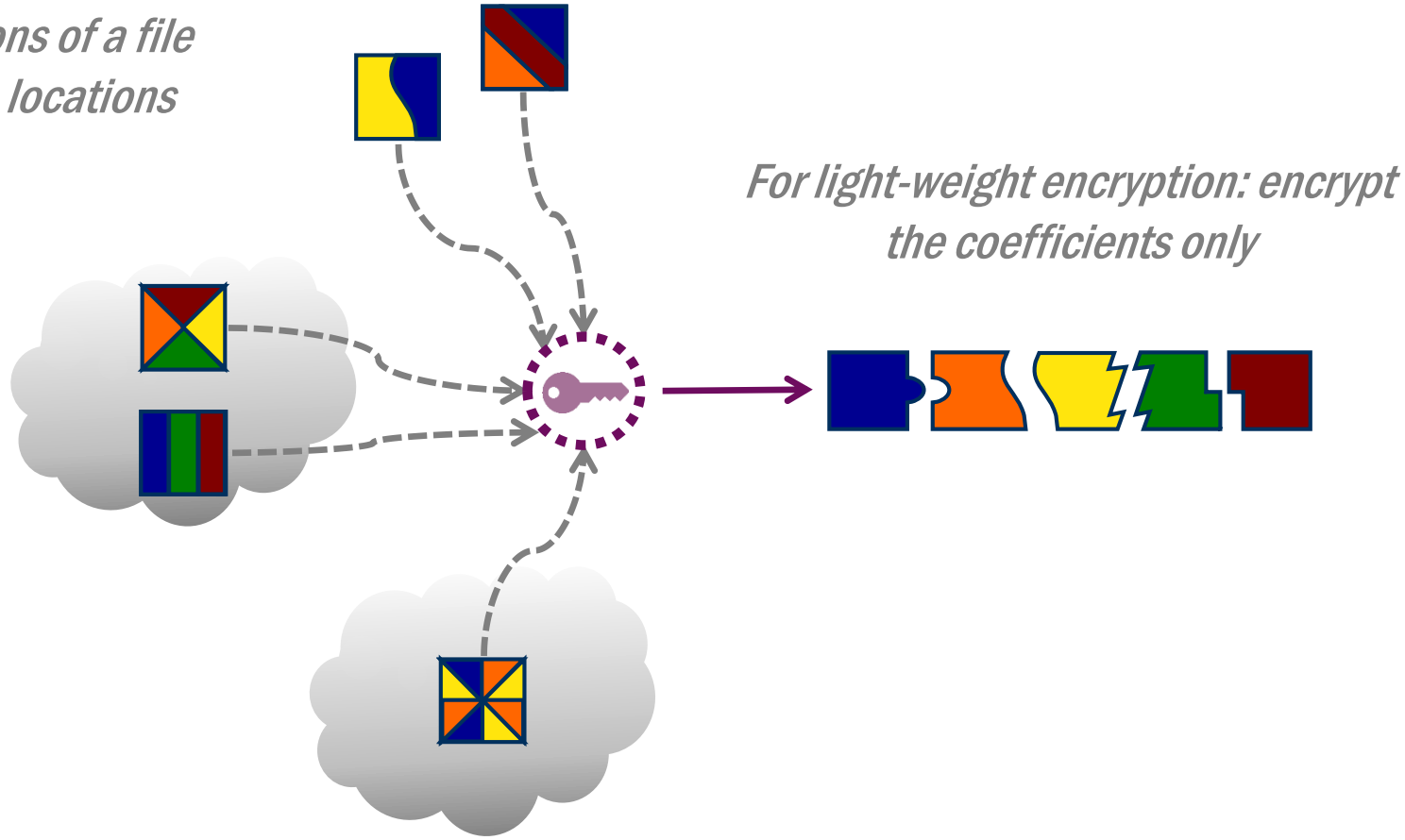
Coding as an Additional Security Measure



- Data on a given path/cloud acts as a cypher

RLNC Improves Cloud Security

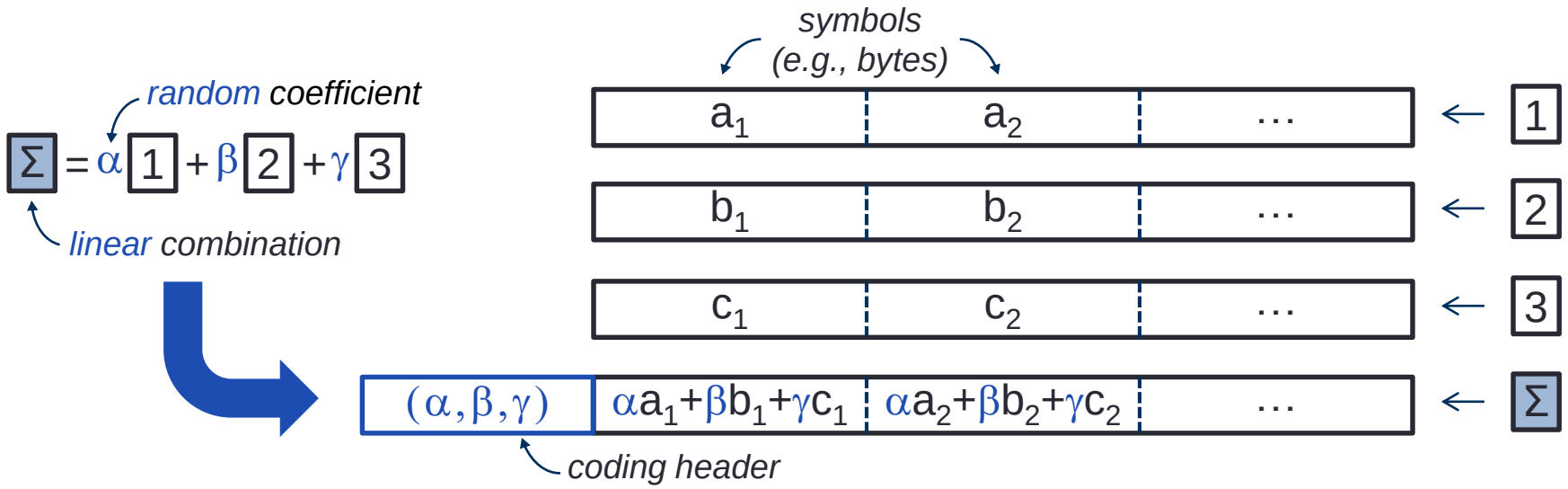
Store portions of a file in multiple locations



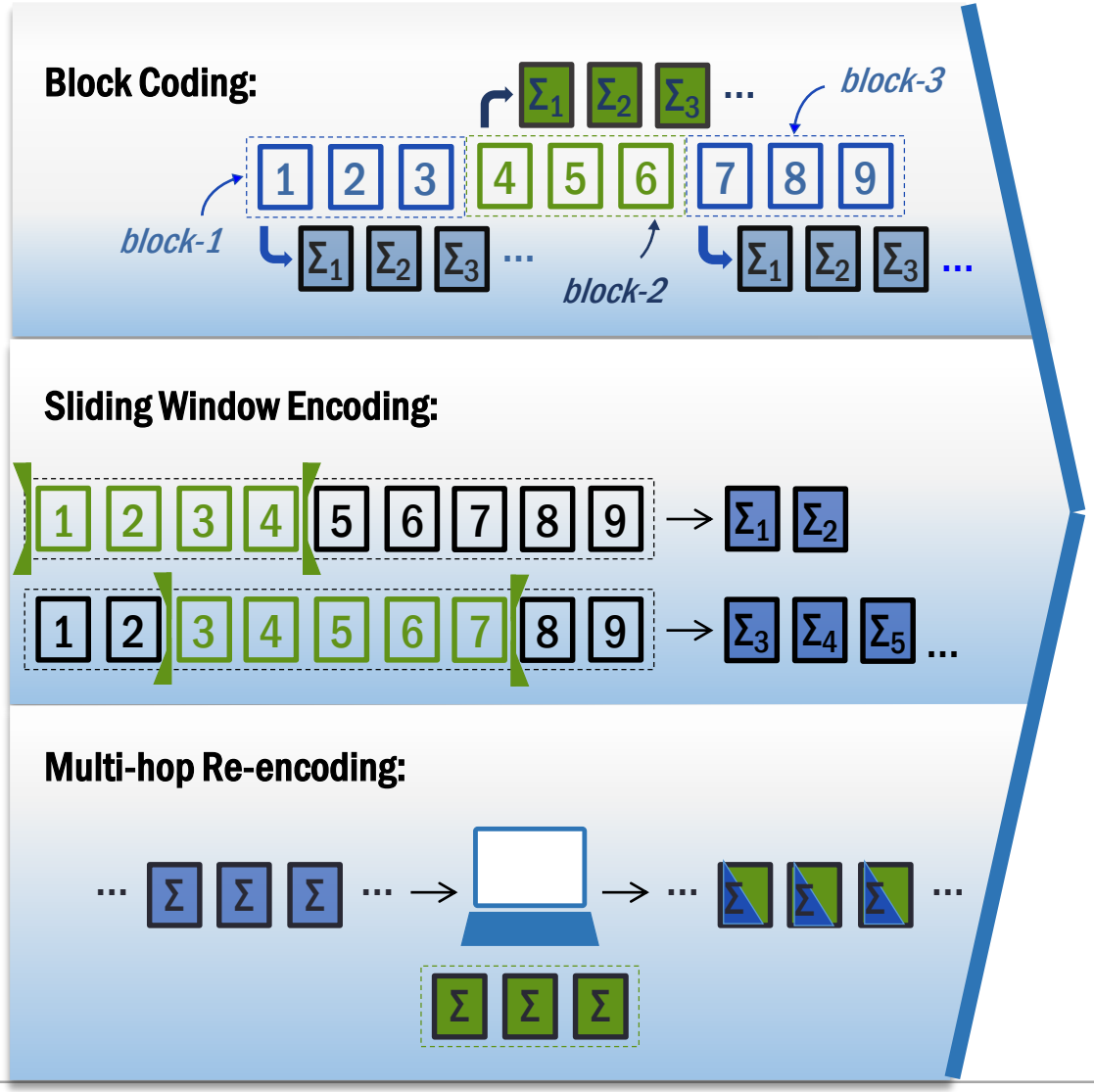
For light-weight encryption: encrypt the coefficients only

Mixture equations "unlock" data from mixtures

How does RLNC work?



- Random Generation of Coefficients
- Code Embedded within Data
- No State Tracking
- ***Versatile*** Code



decoding scheme

(simple equation-solving)

$$1 = \alpha'_1 \Sigma_1 + \beta'_1 \Sigma_2 + \gamma'_1 \Sigma_3$$

$$2 = \alpha'_2 \Sigma_1 + \beta'_2 \Sigma_2 + \gamma'_2 \Sigma_3$$

$$3 = 1 + \beta'_3 \Sigma_2 + \gamma'_3 \Sigma_3$$

obtained through Gaussian Elimination

Can decode using both encoded and un-encoded packets

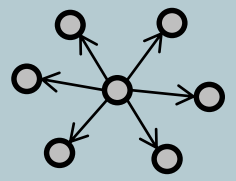
RLNC: The Technology

Coding Today

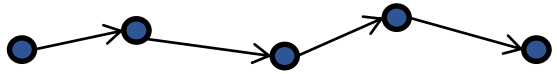
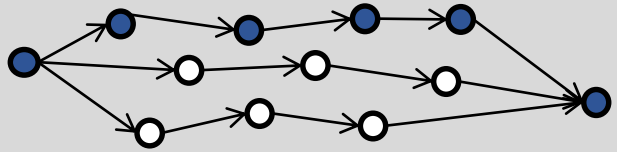
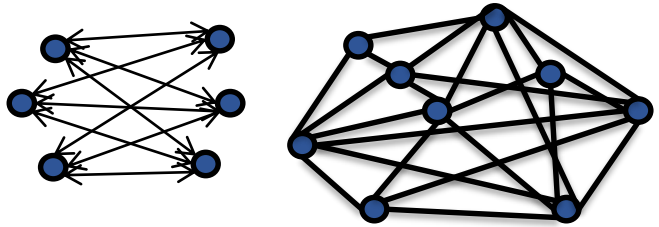
(all End-to-End)

Classical

Multicast



Coding Tomorrow with RLNC

Classical + Sliding Window Encoding 	Real time video streaming, TCP, SDN…
Multihop 	Edge caches, wireless mesh, reliable multicast, satellites, small relay topologies, SDN…
Multipath 	Multi-source streaming Multipath TCP, channel bundling, heterogeneous network combining, SDN…
Multisource – Multi-destination / Mesh 	Distributed cloud, SDN, advanced mesh (IoT, car2car, M2M, smart grid) …

Key Parameters of RLNC

- Generation size: number of packets that are (currently) coded together.
- Field size: number of elements in the finite field

- Both have an impact on:
  - Performance
  - Complexity

Field Size

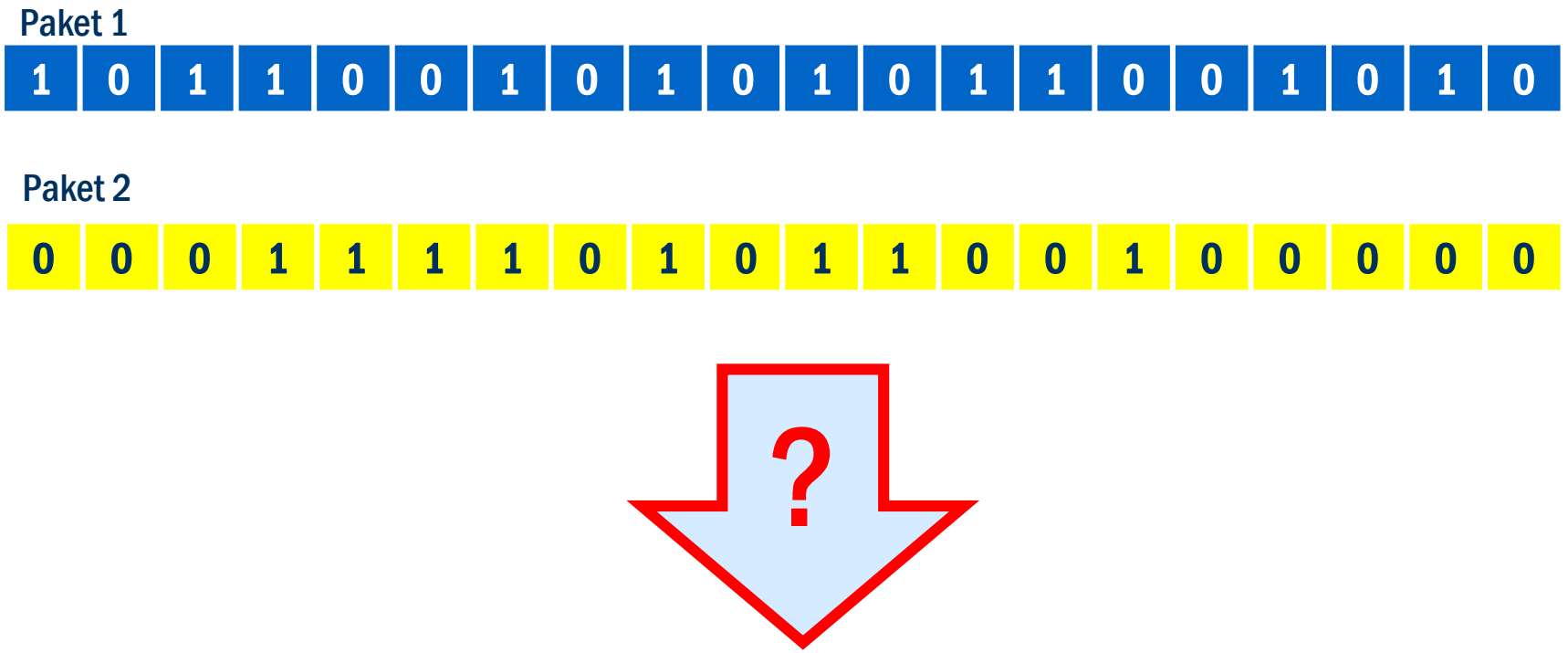
Paket 1

1	0	1	1	0	0	1	0	1	0	1	0	1	1	0	0	1	0	1	0

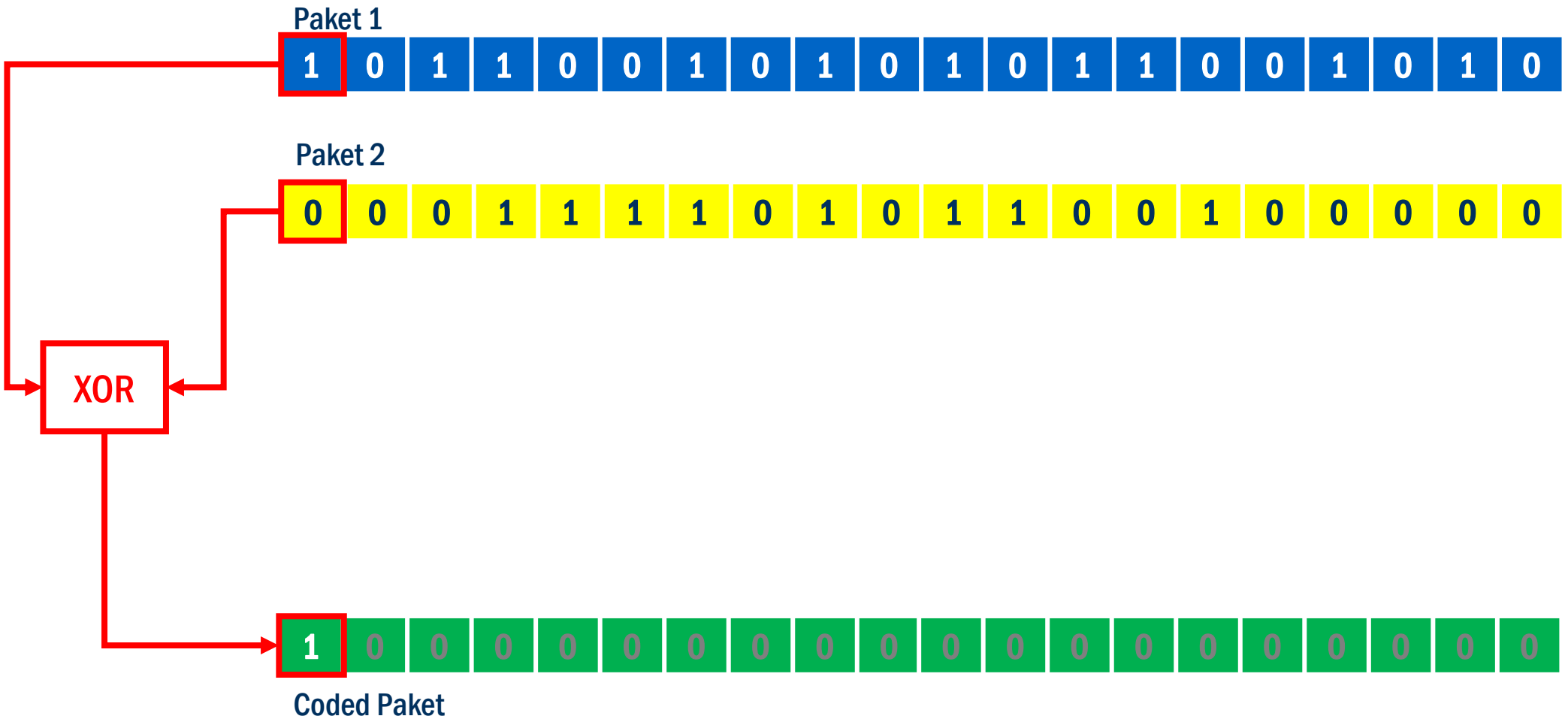
Paket 2

0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0

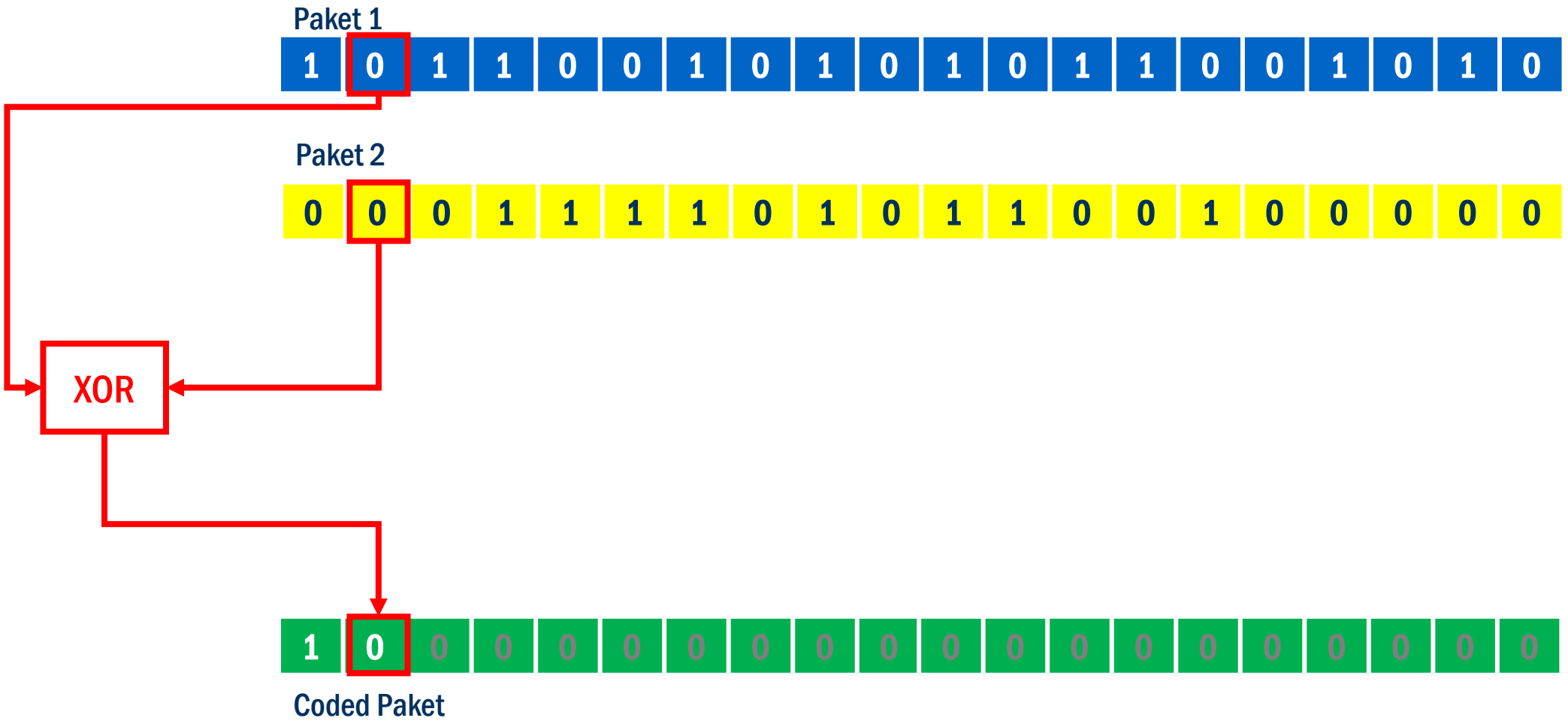
Field Size


Paket 1
1 0 1 1 0 0 1 0 1 0 1 0 1 1 0 0 1 0 1 0
Paket 2
0 0 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0 0 0
?
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Coded Paket (initially empty)

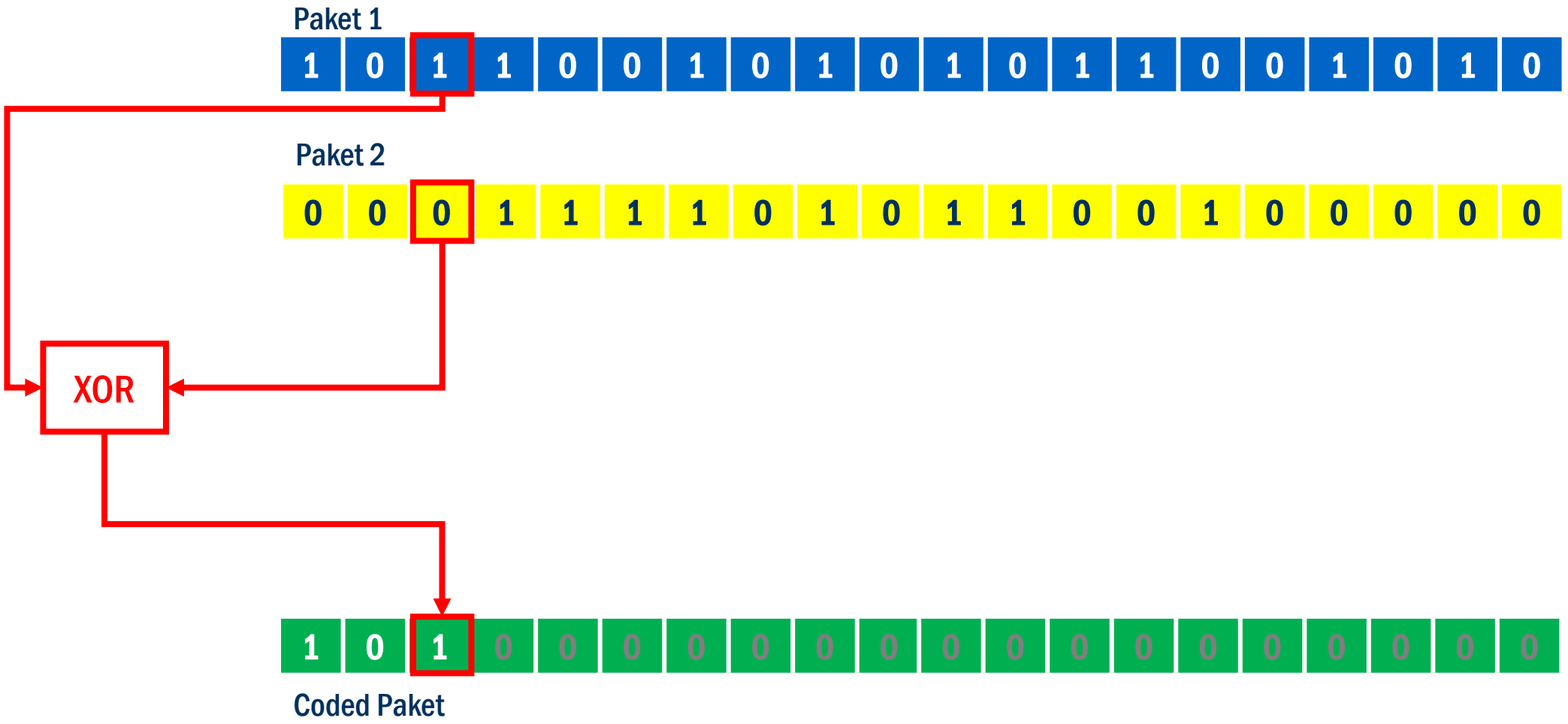
Field Size GF(2)



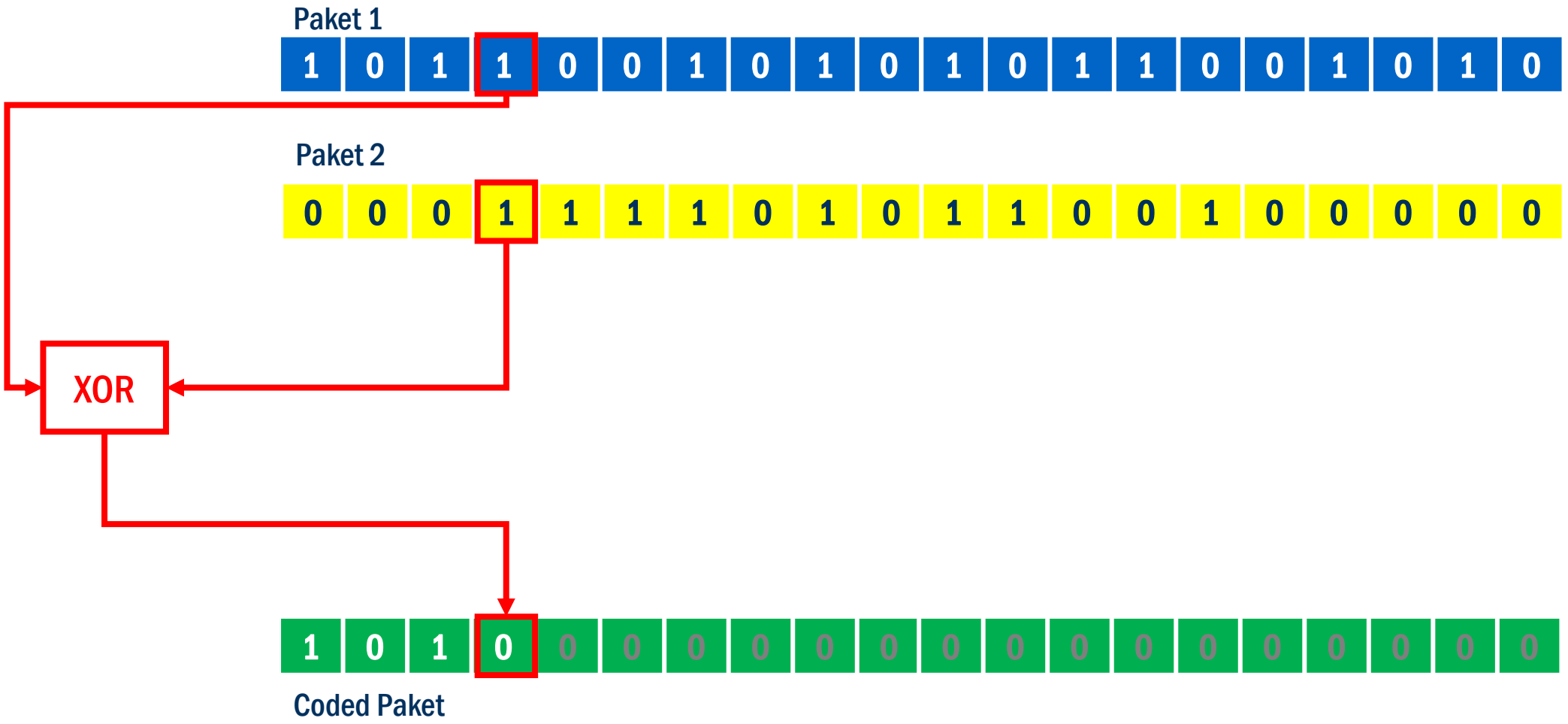
Field Size GF(2)



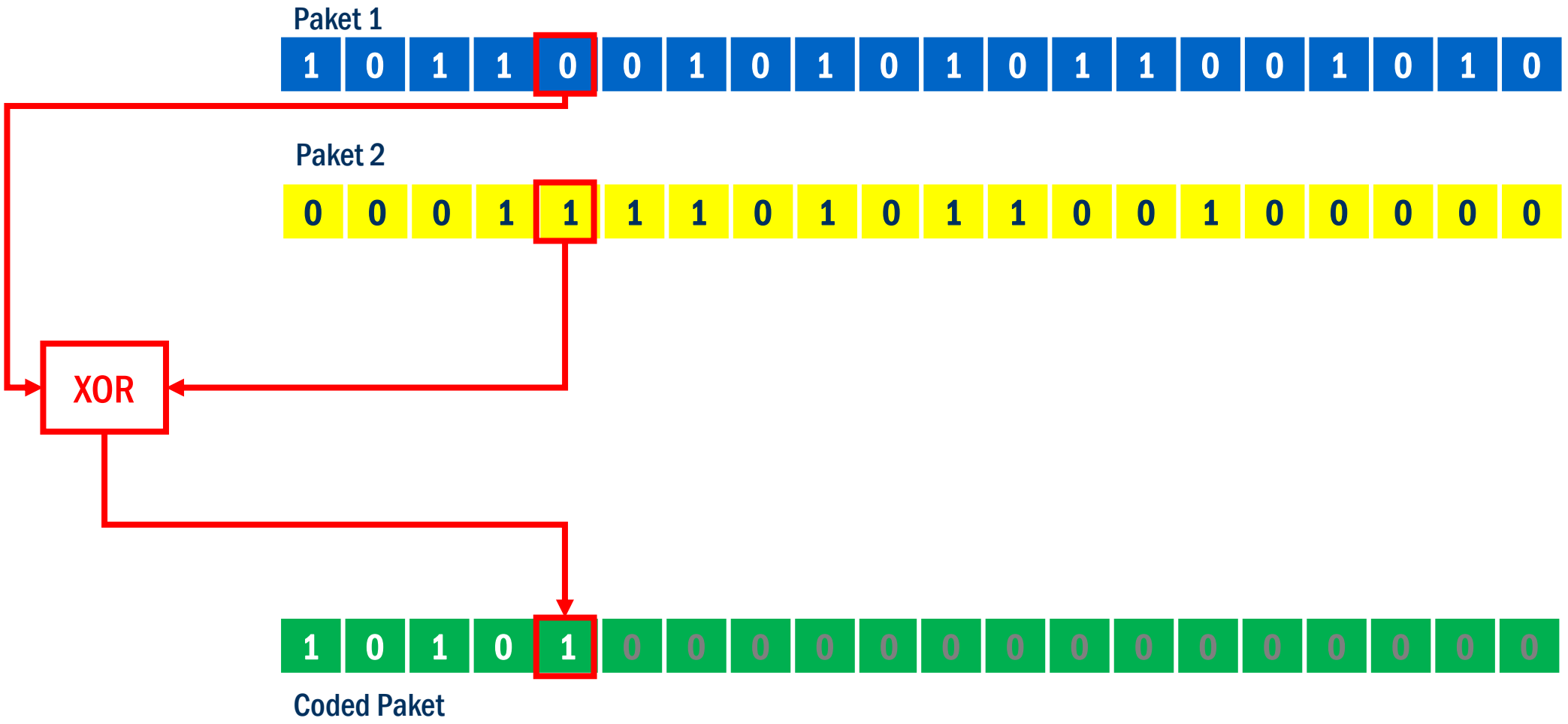
Field Size GF(2)



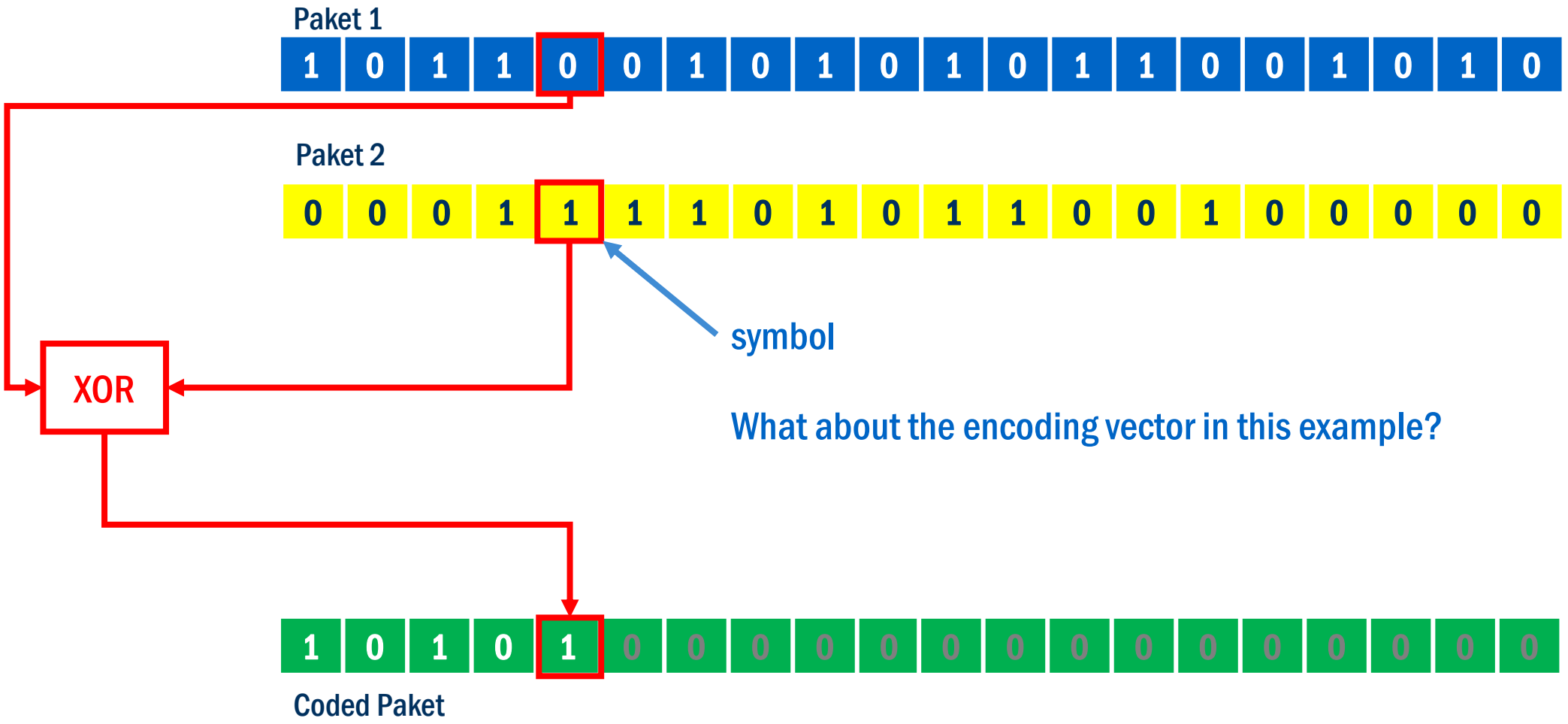
Field Size GF(2)



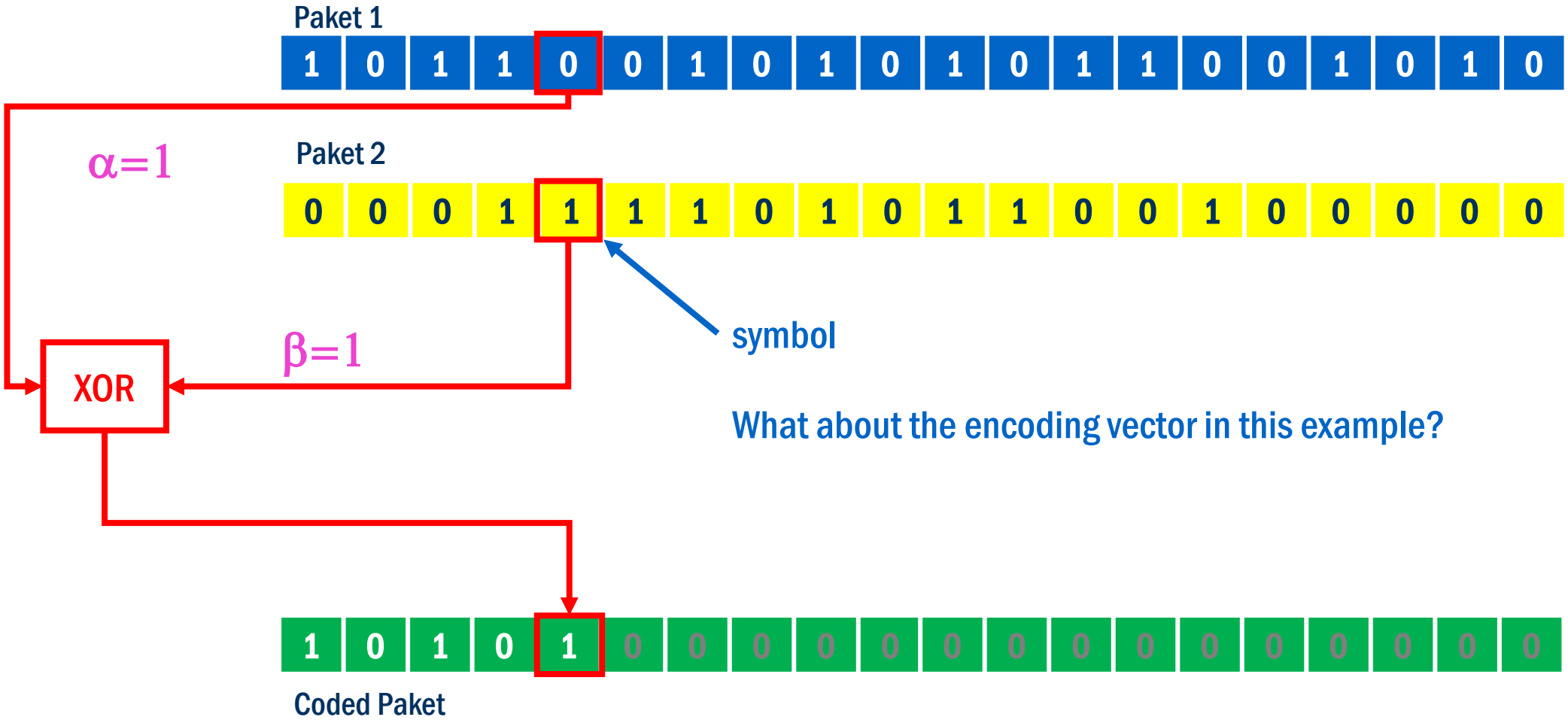
Field Size GF(2)



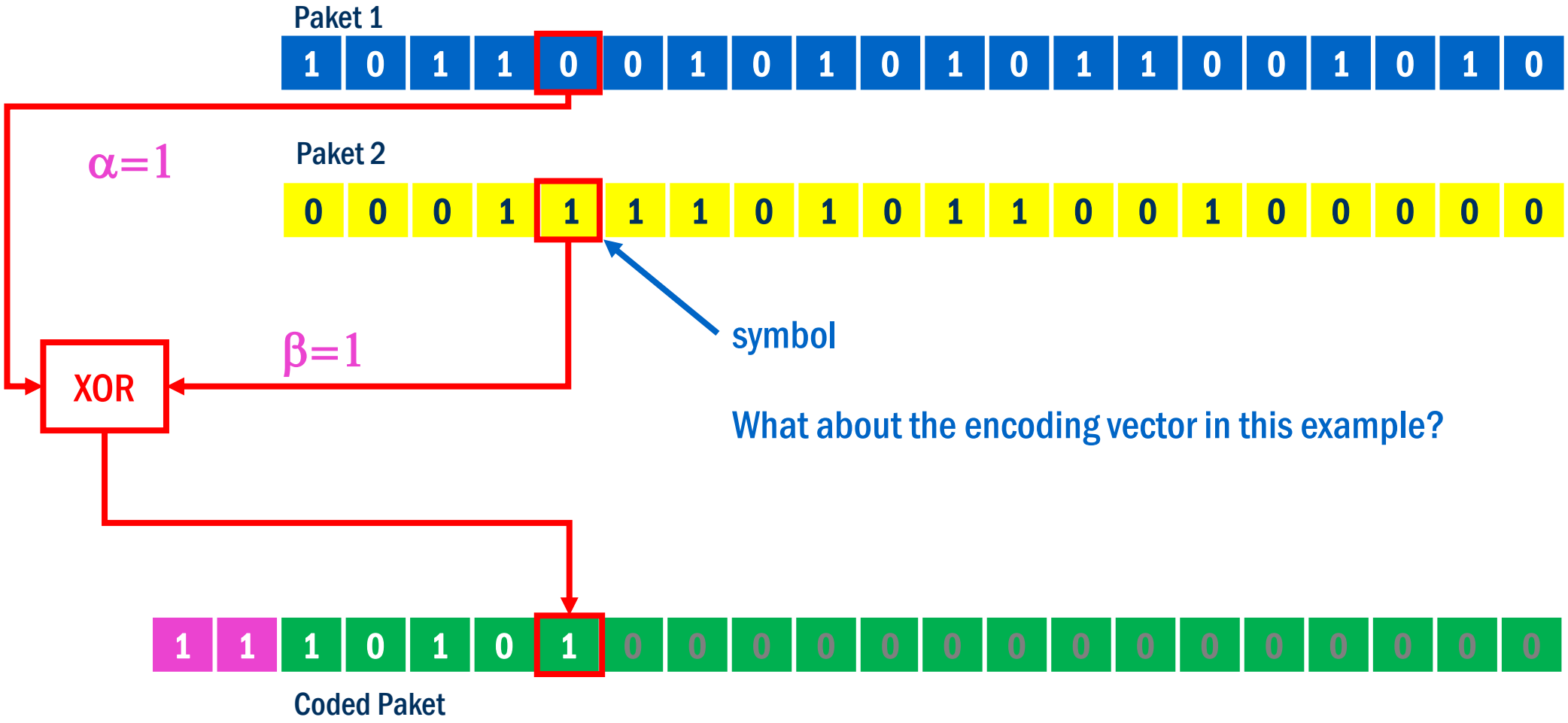
Field Size GF(2)



Field Size GF(2)

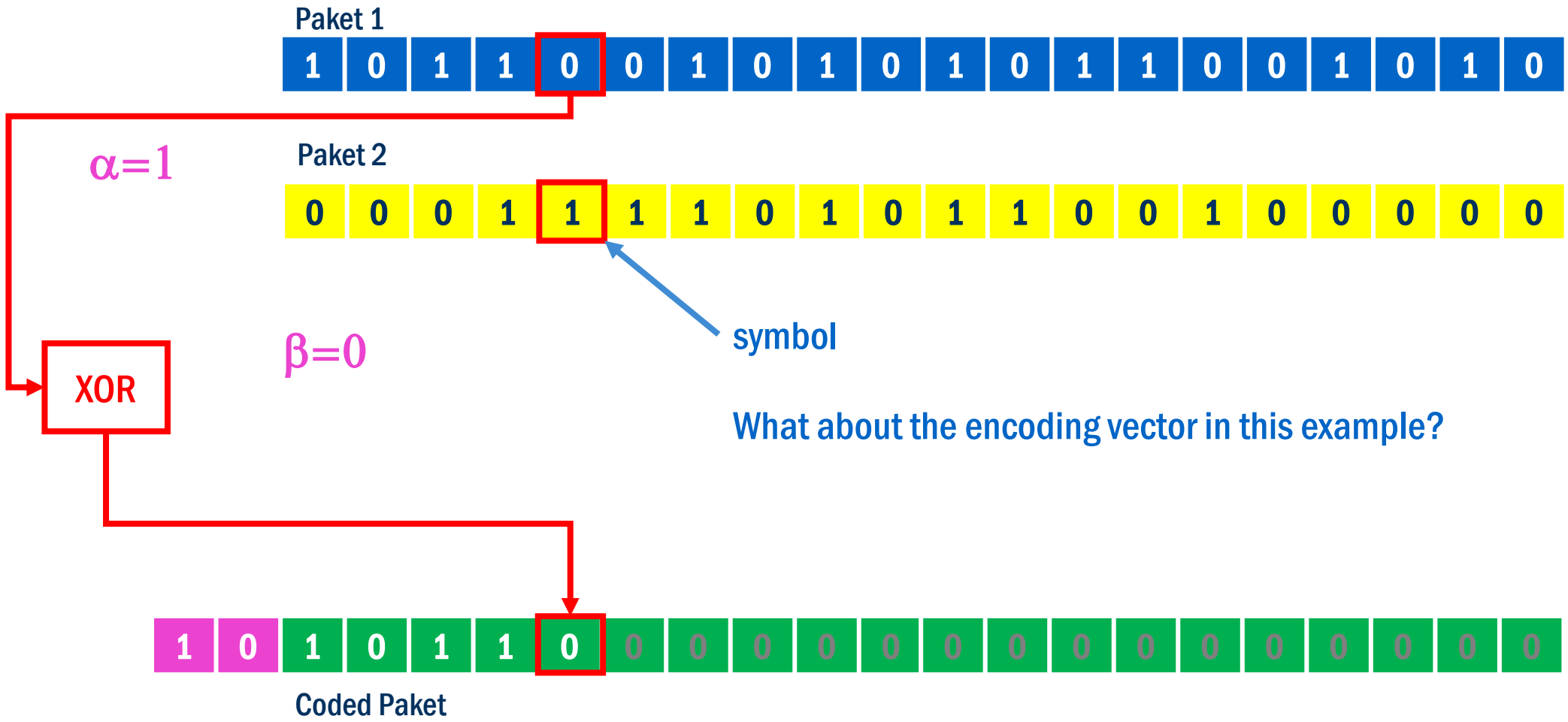


Field Size GF(2)

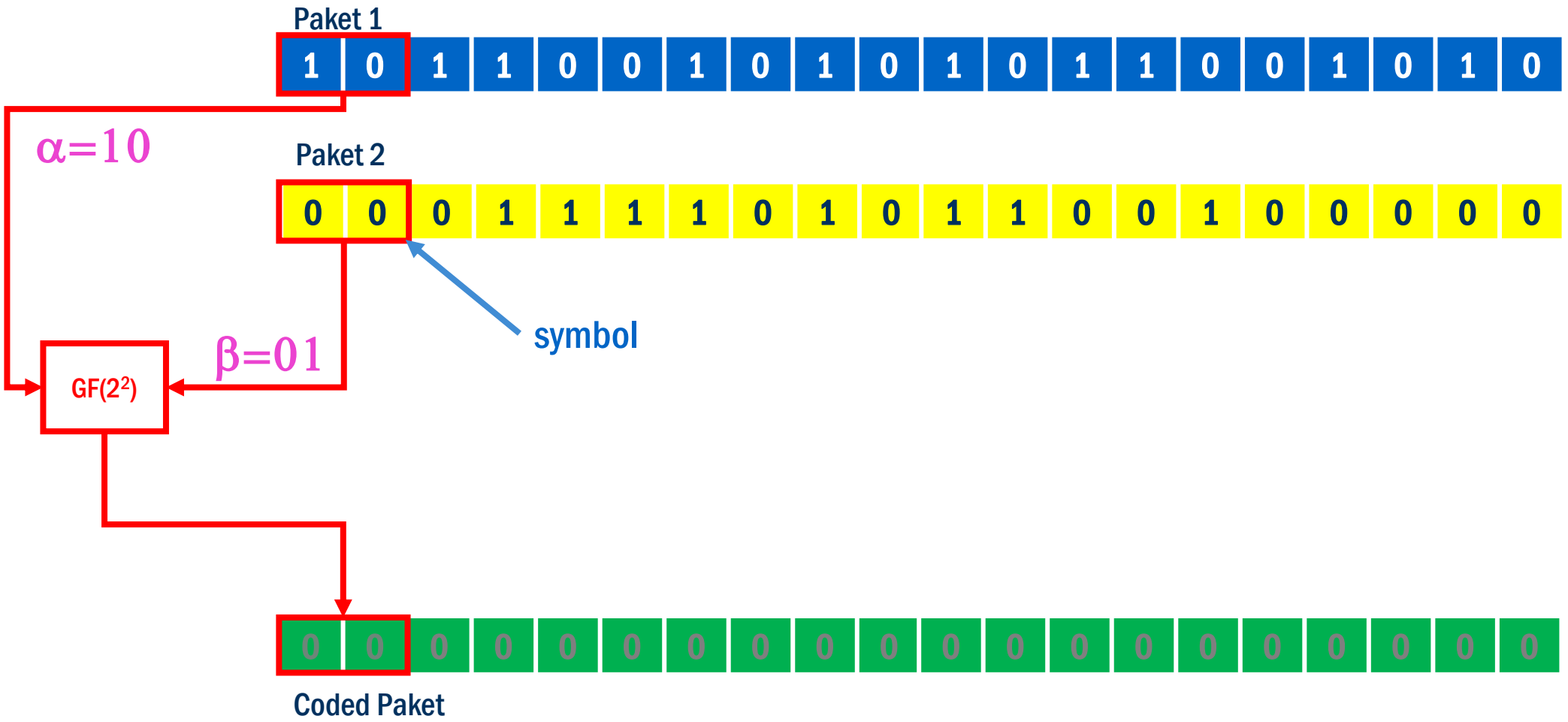


What about the encoding vector in this example?

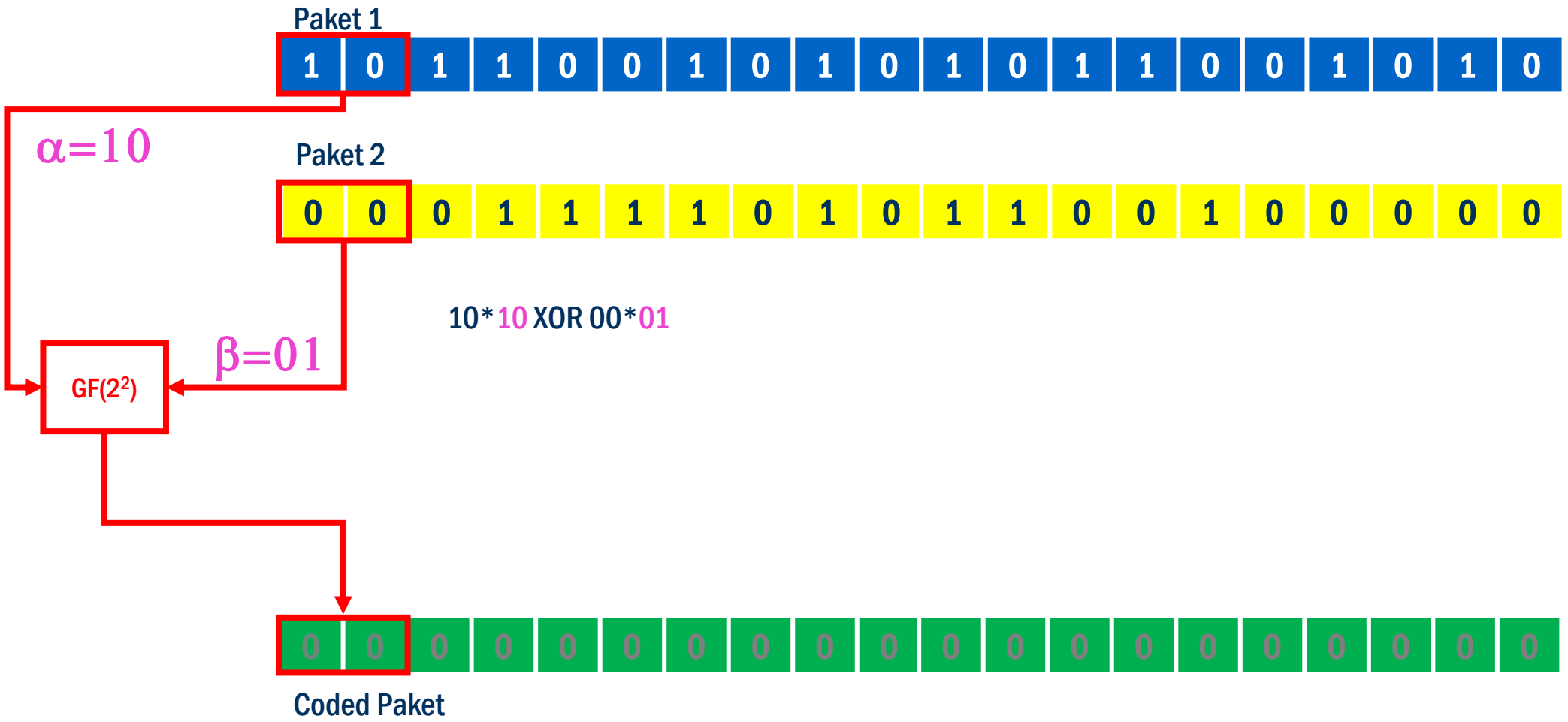
Field Size GF(2)



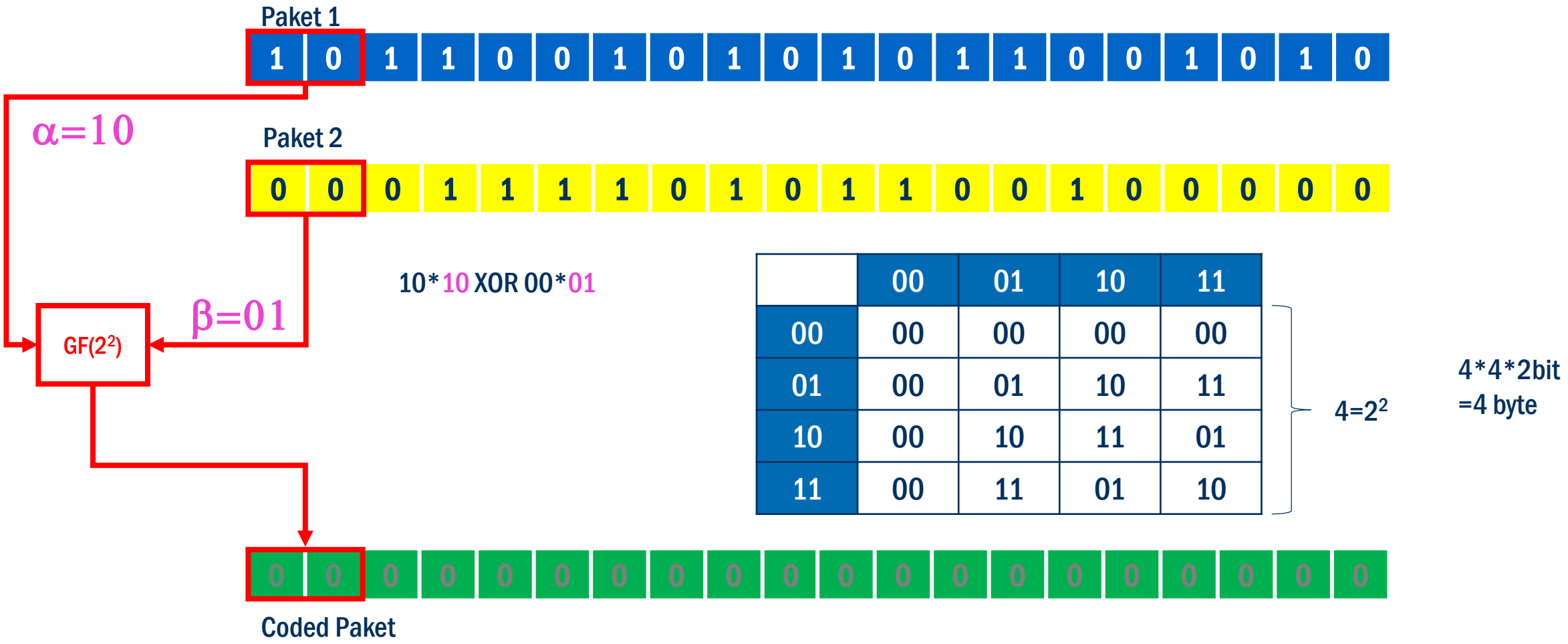
Field Size GF(2^2)



Field Size GF(2^2)



Field Size GF(2^2)

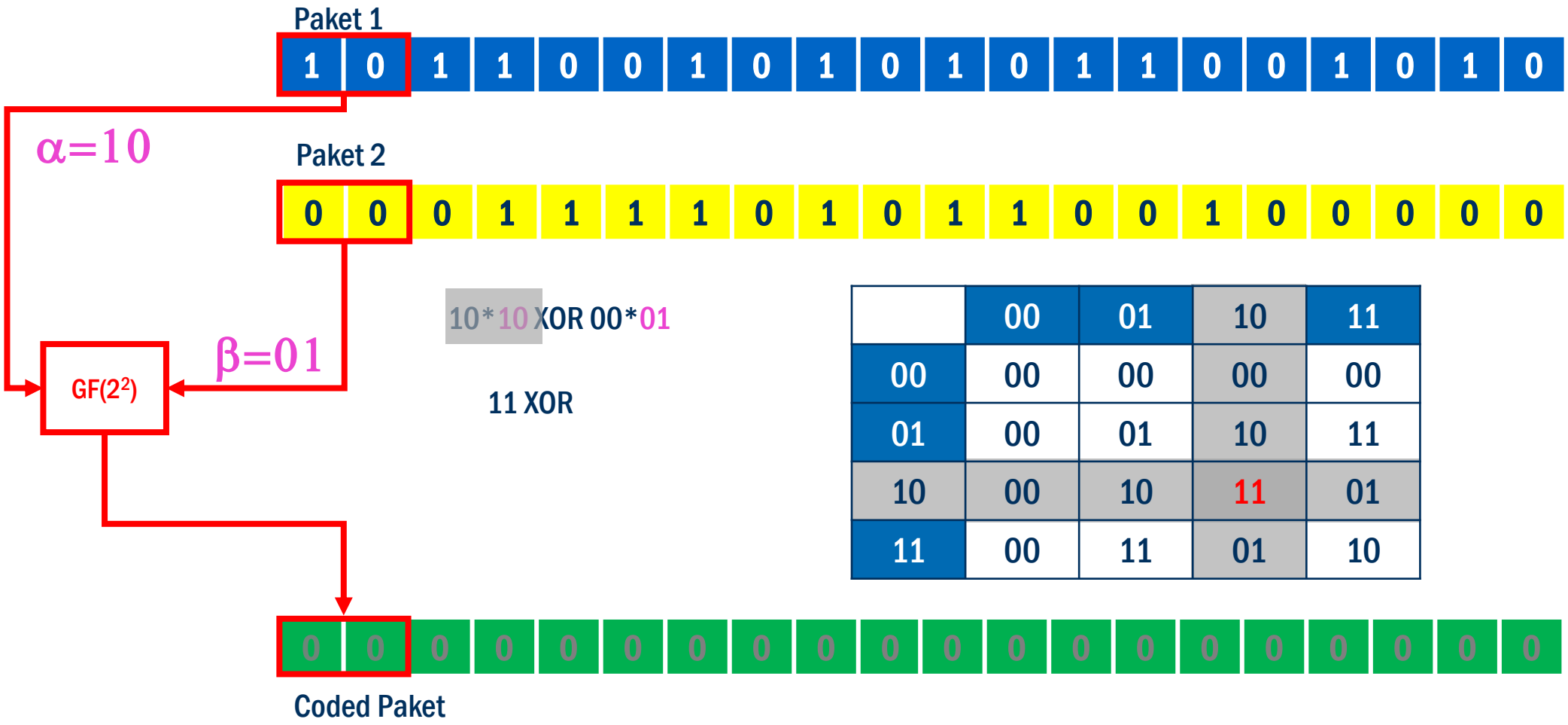


	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

$4=2^2$

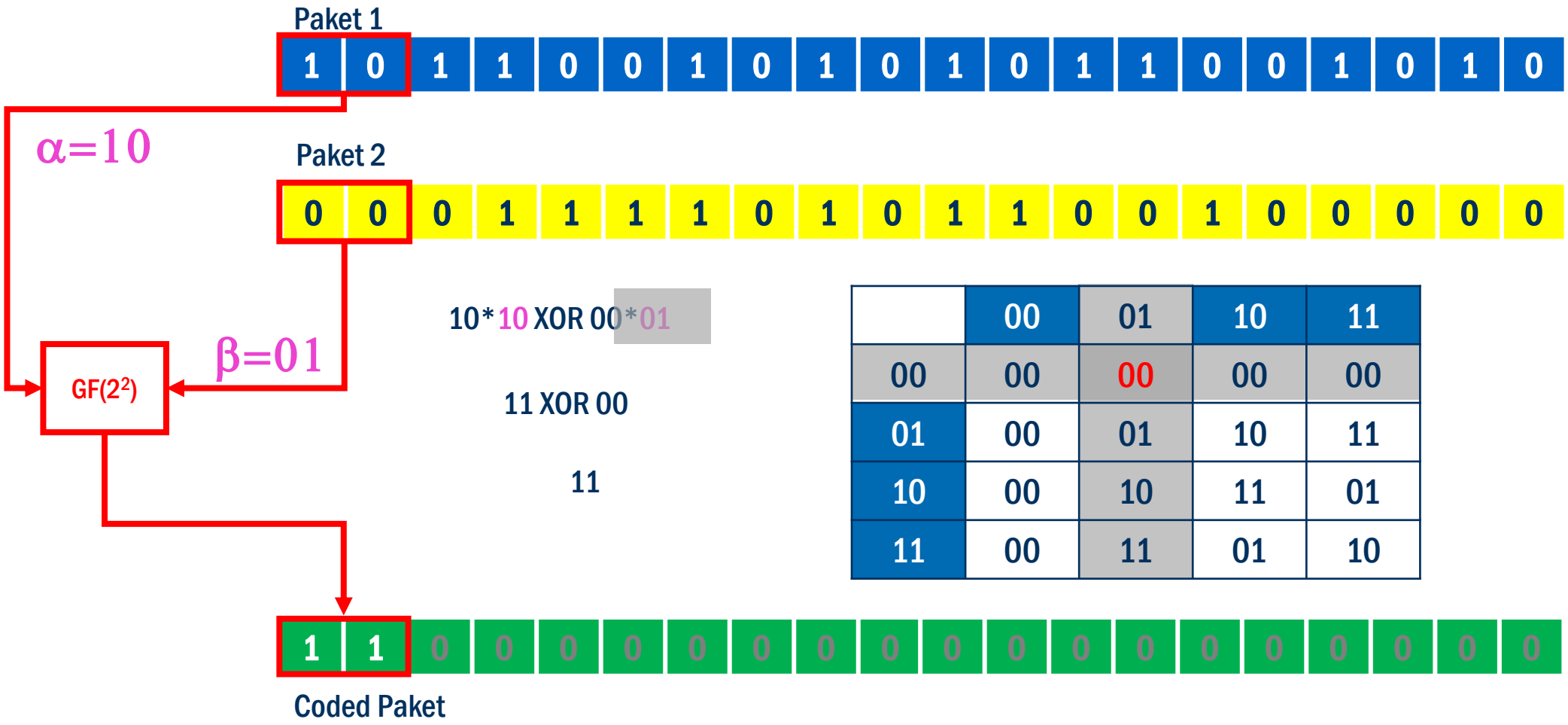
4*4*2bit
=4 byte

Field Size GF(2^2)



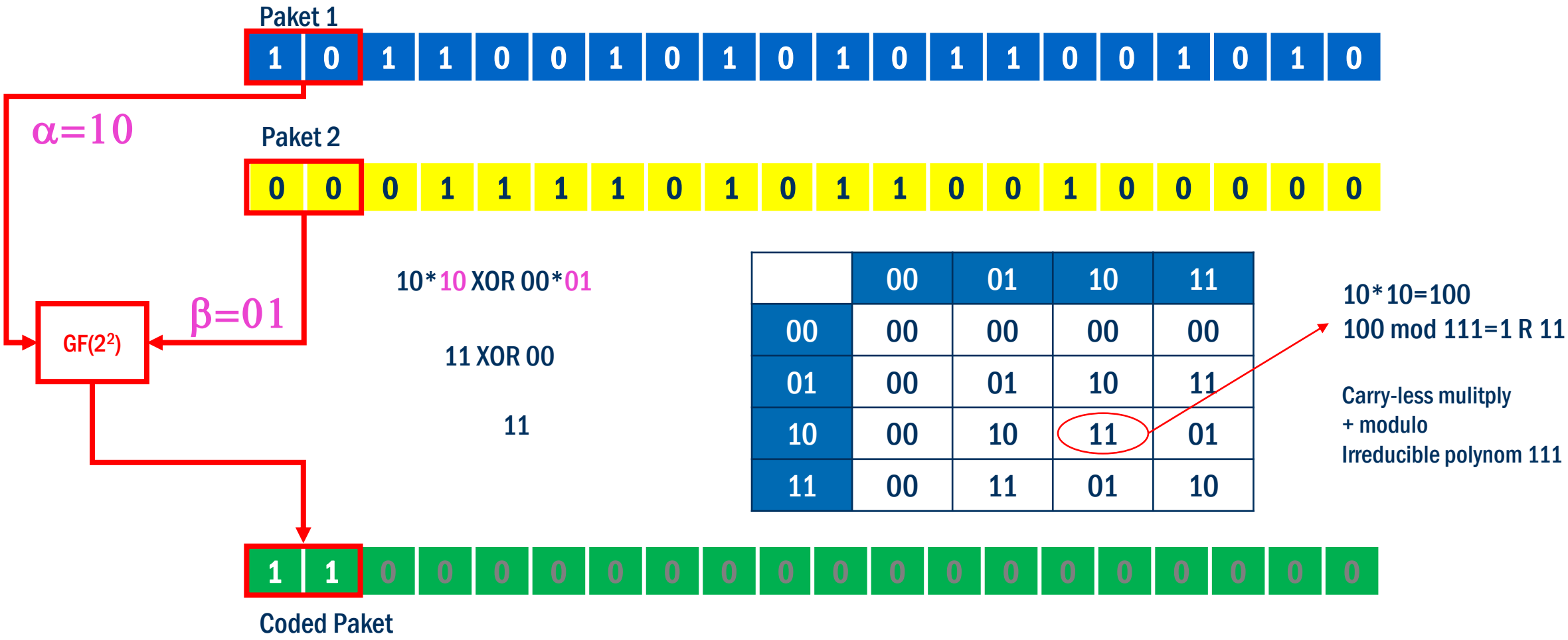
	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

Field Size GF(2^2)



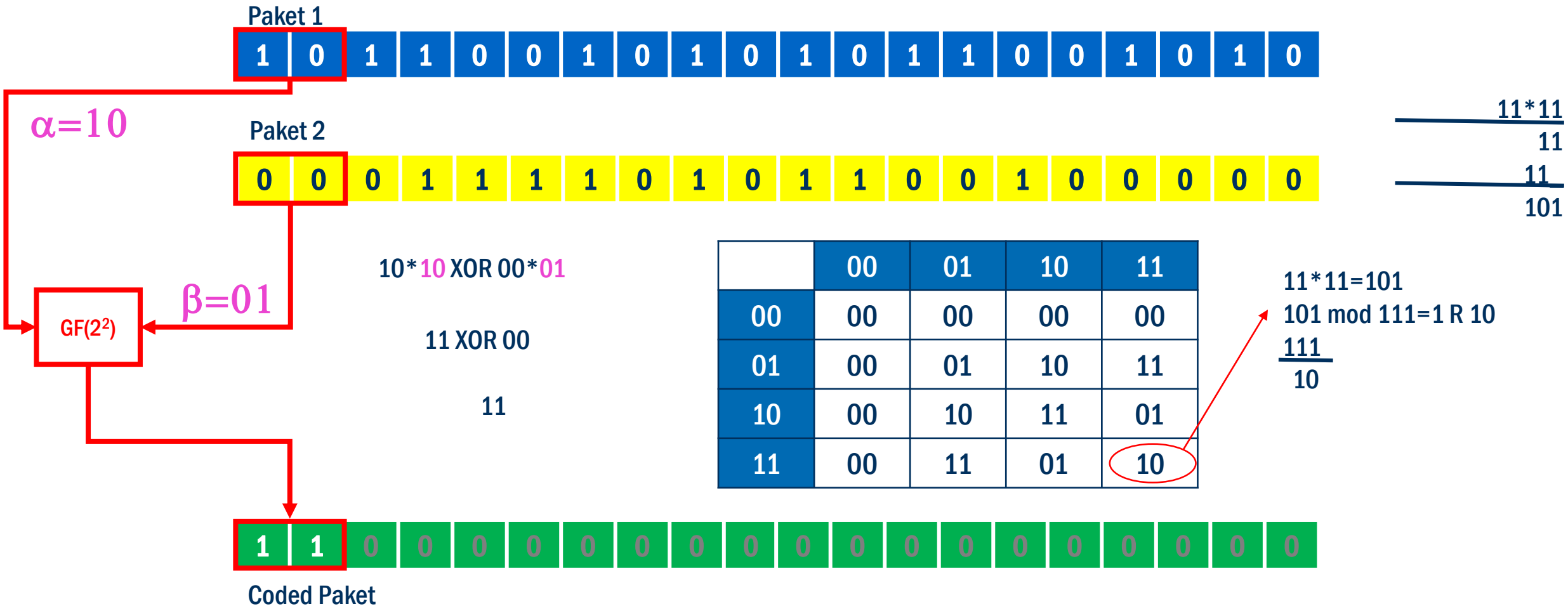
	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

Field Size GF(2^2)



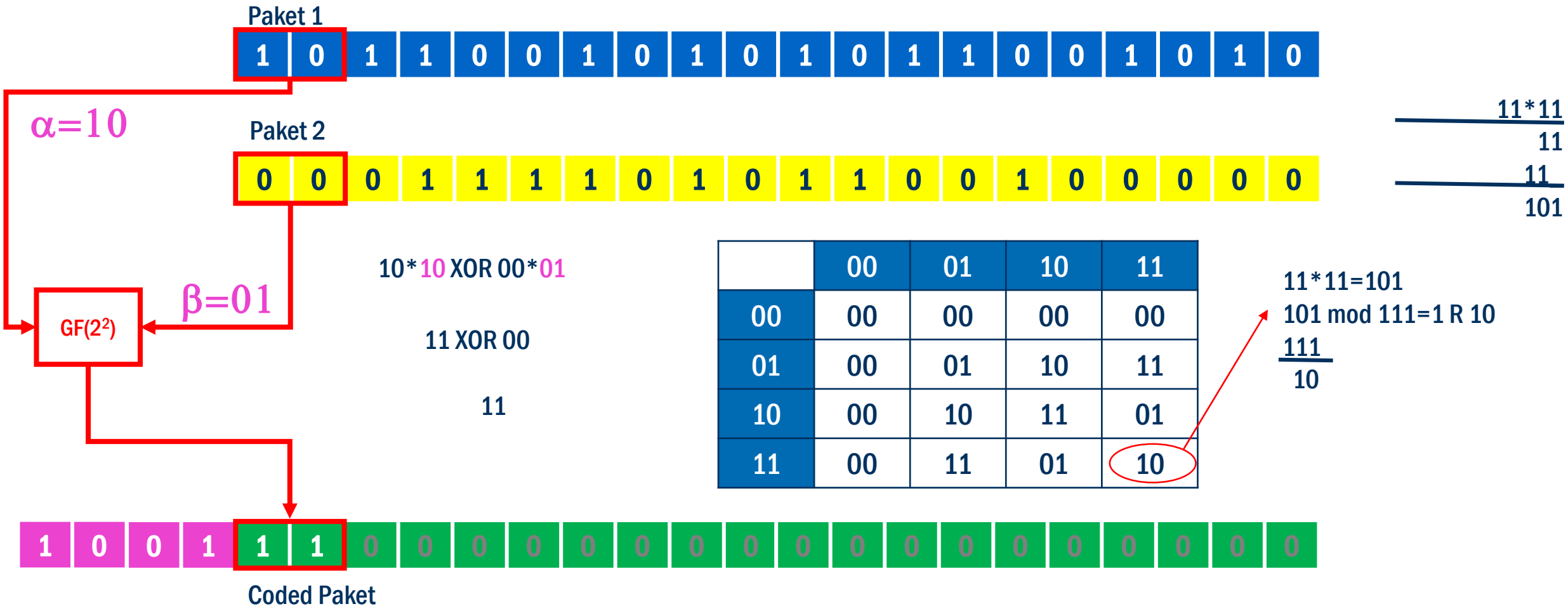
	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

Field Size GF(2^2)



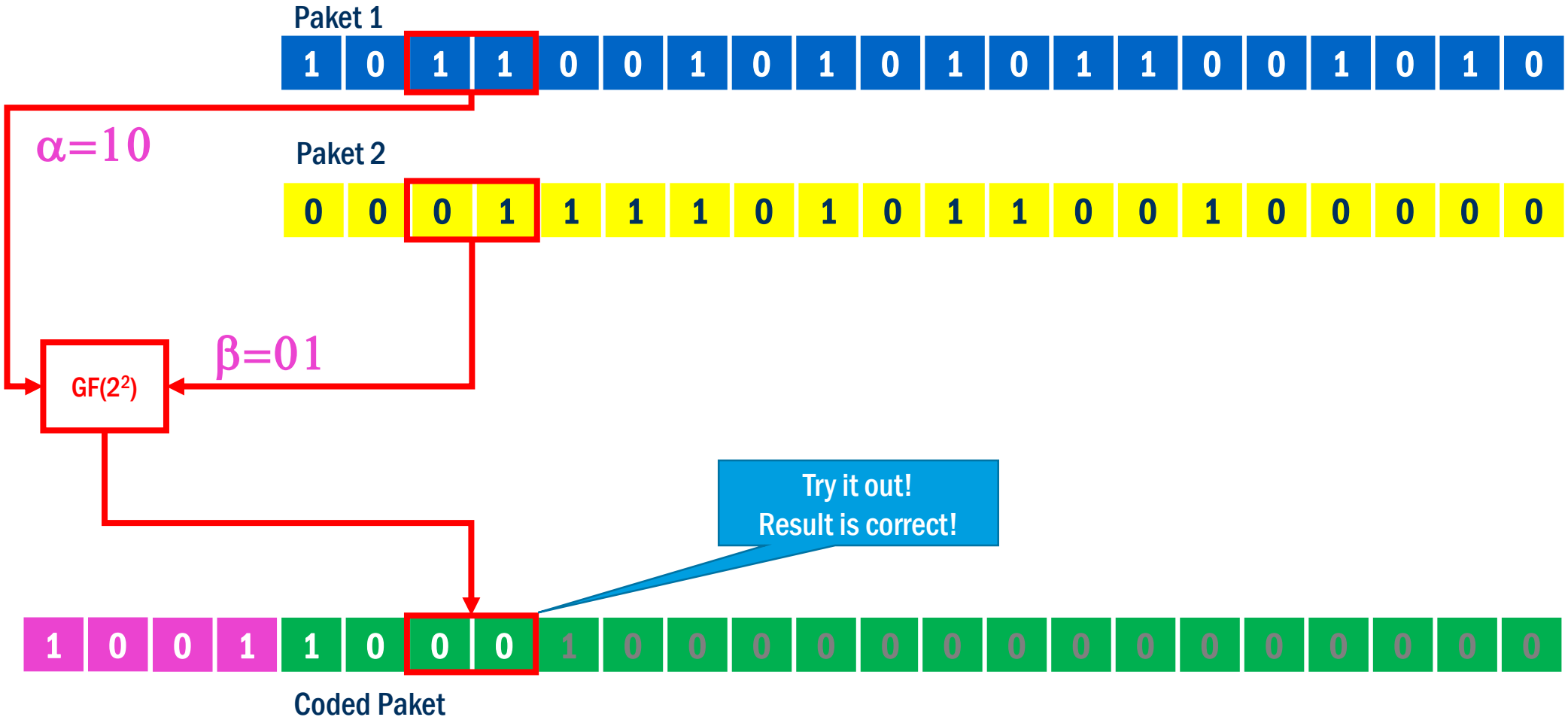
	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

Field Size GF(2^2)

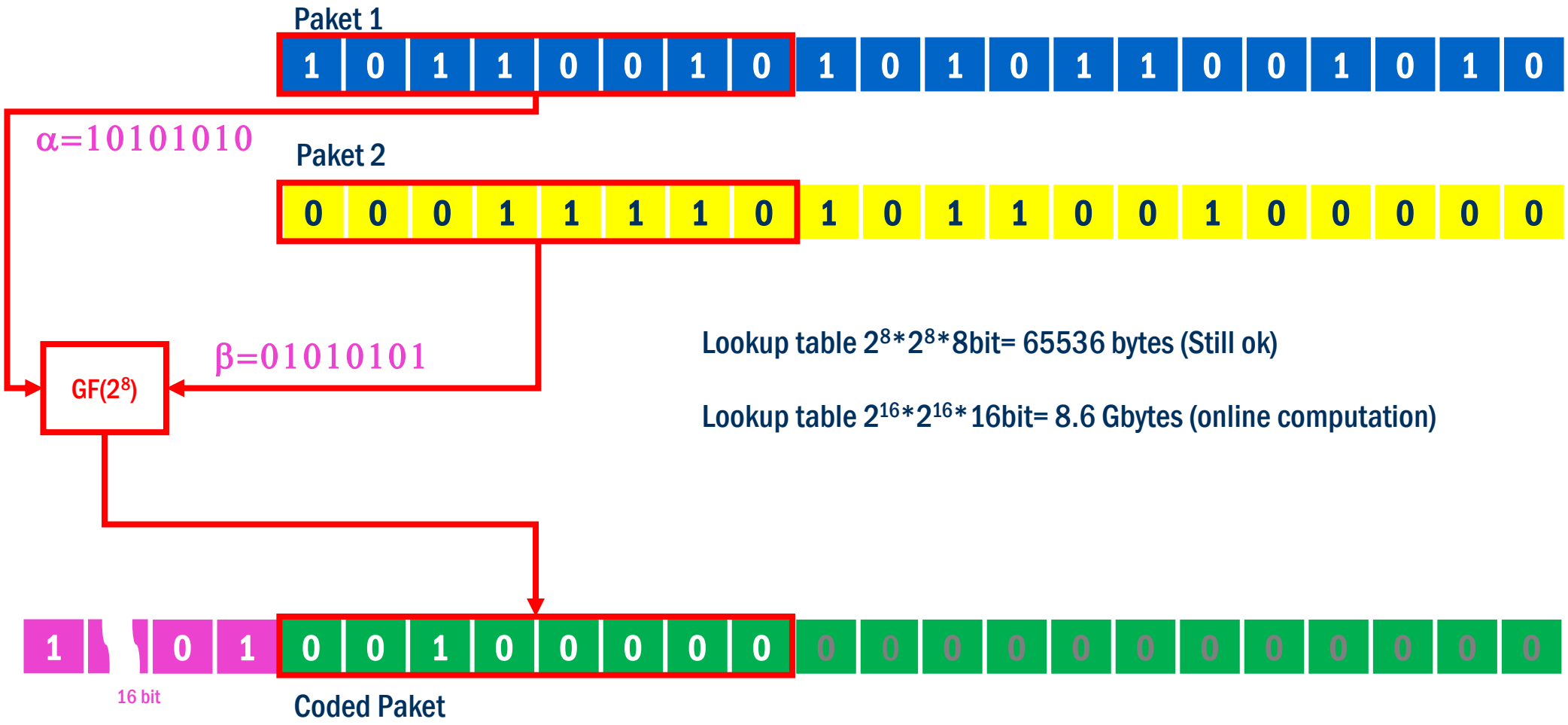


	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

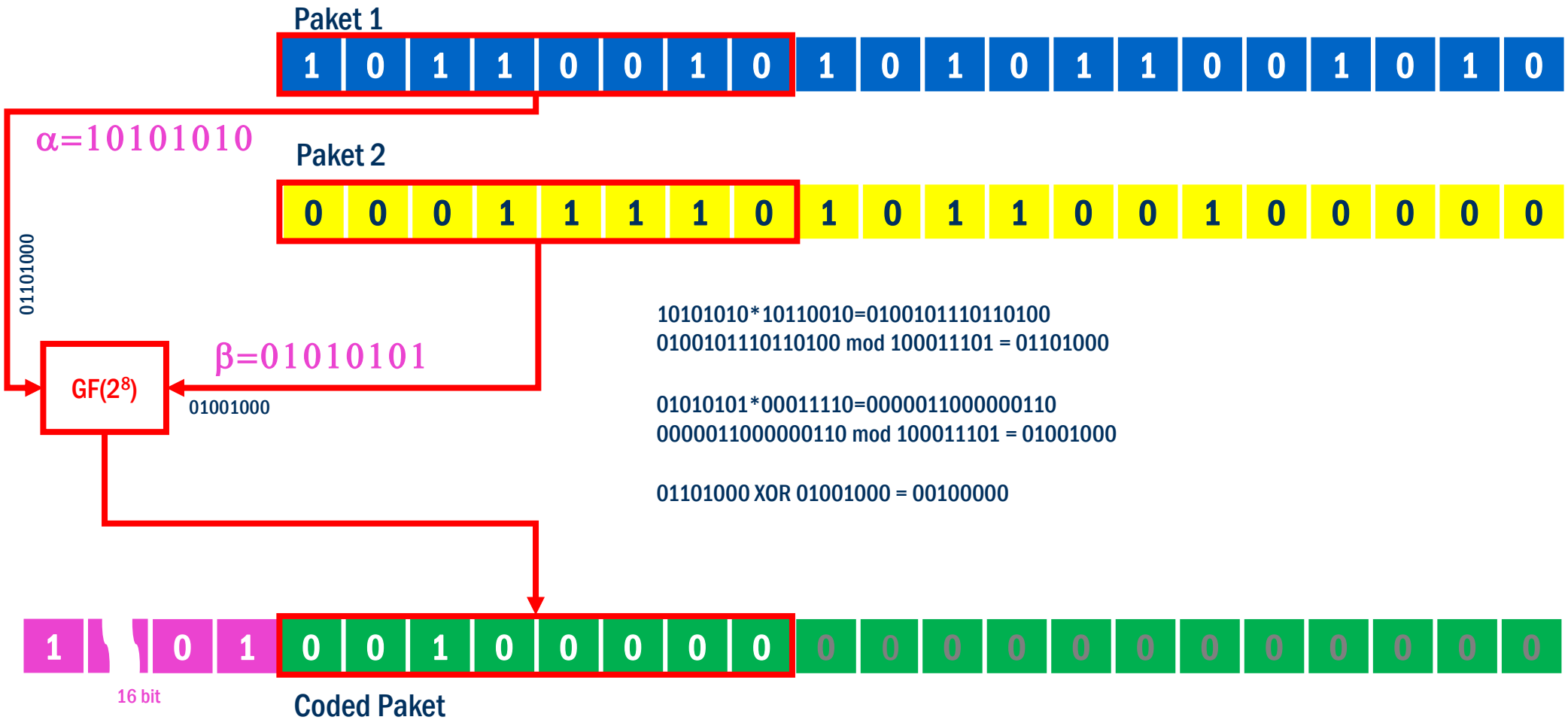
Field Size GF(2^2)



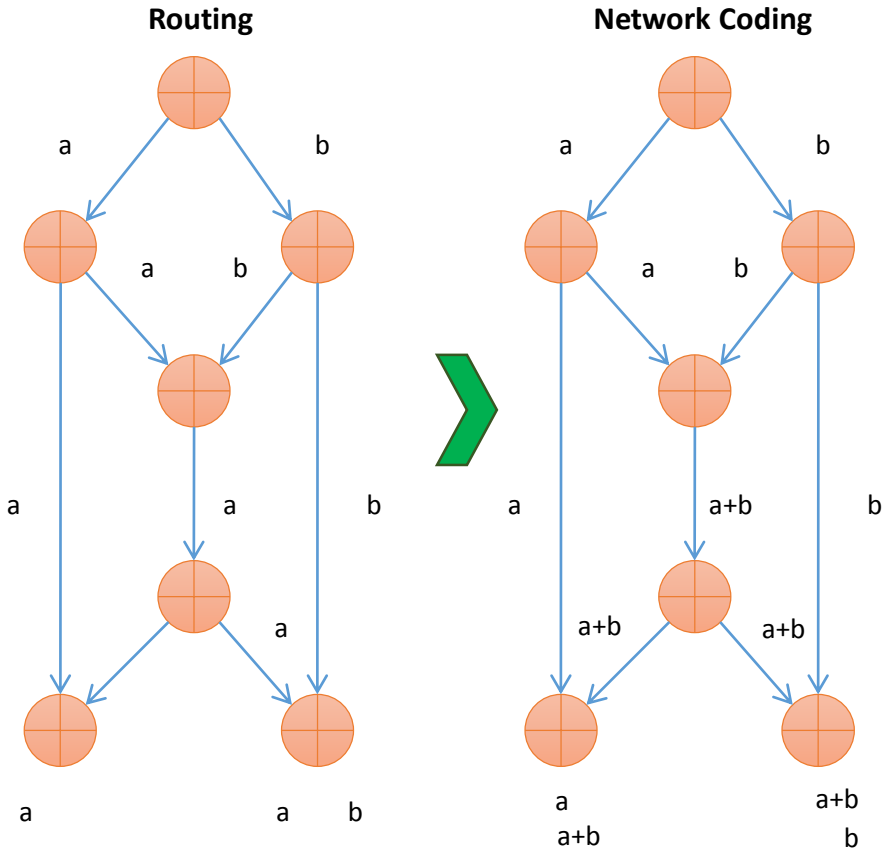
Field Size GF(2^8)



Field Size GF(2^8)



Network Coding



a

Rate: 1.5 symbols/time
Distributed (but planned)
Sub-optimal
Low processing cost

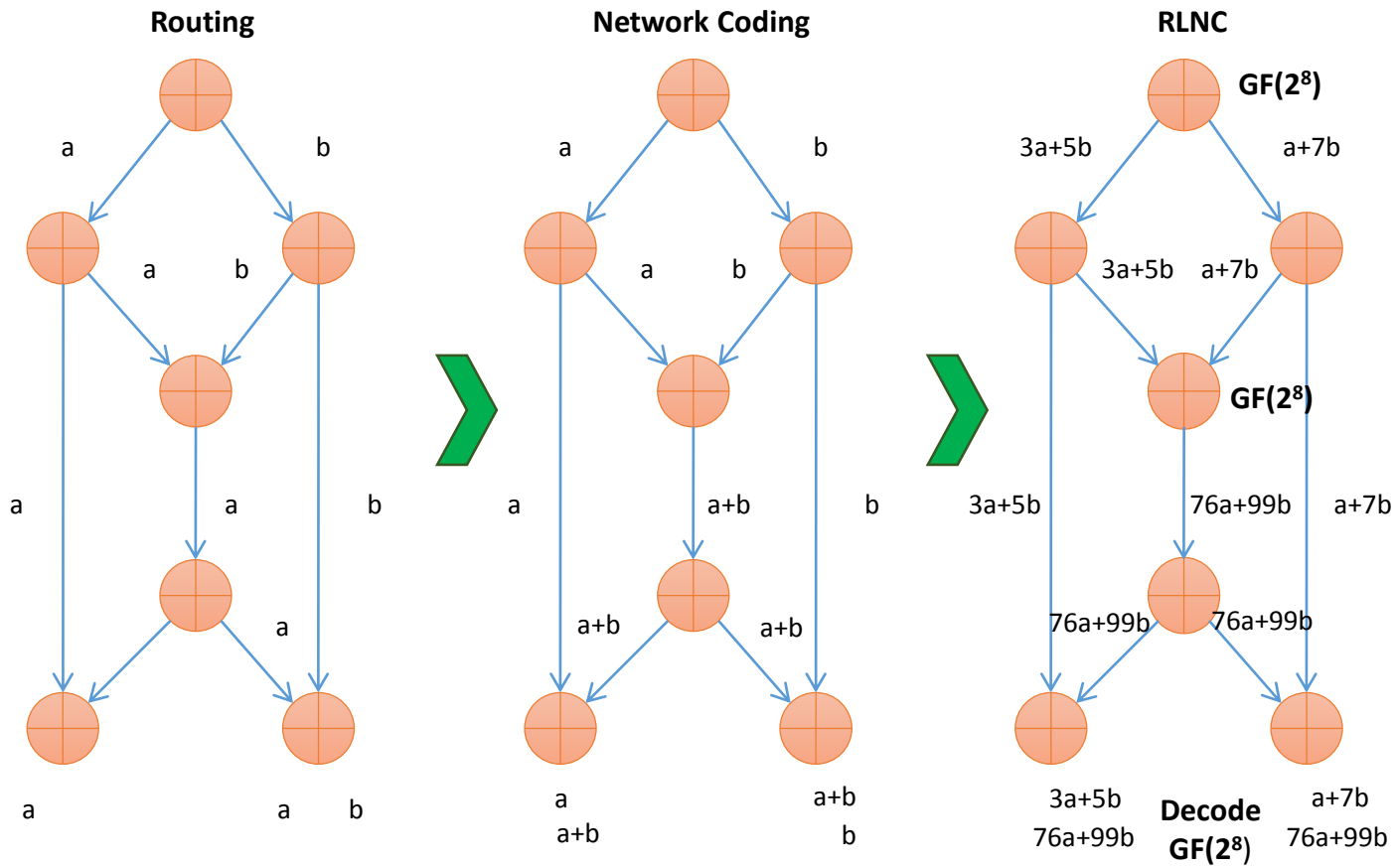
a b

a b

Rate: 2 symbols/time
Centralized, Planned
Optimal
Low-Medium processing cost
One Finite Field in use
Does not consider device capabilities

- One encoder, one decoder
- One recoder

Network Coding



a | a b

Rate: 1.5 symbols/time
Distributed (but planned)
Sub-optimal
Low processing cost

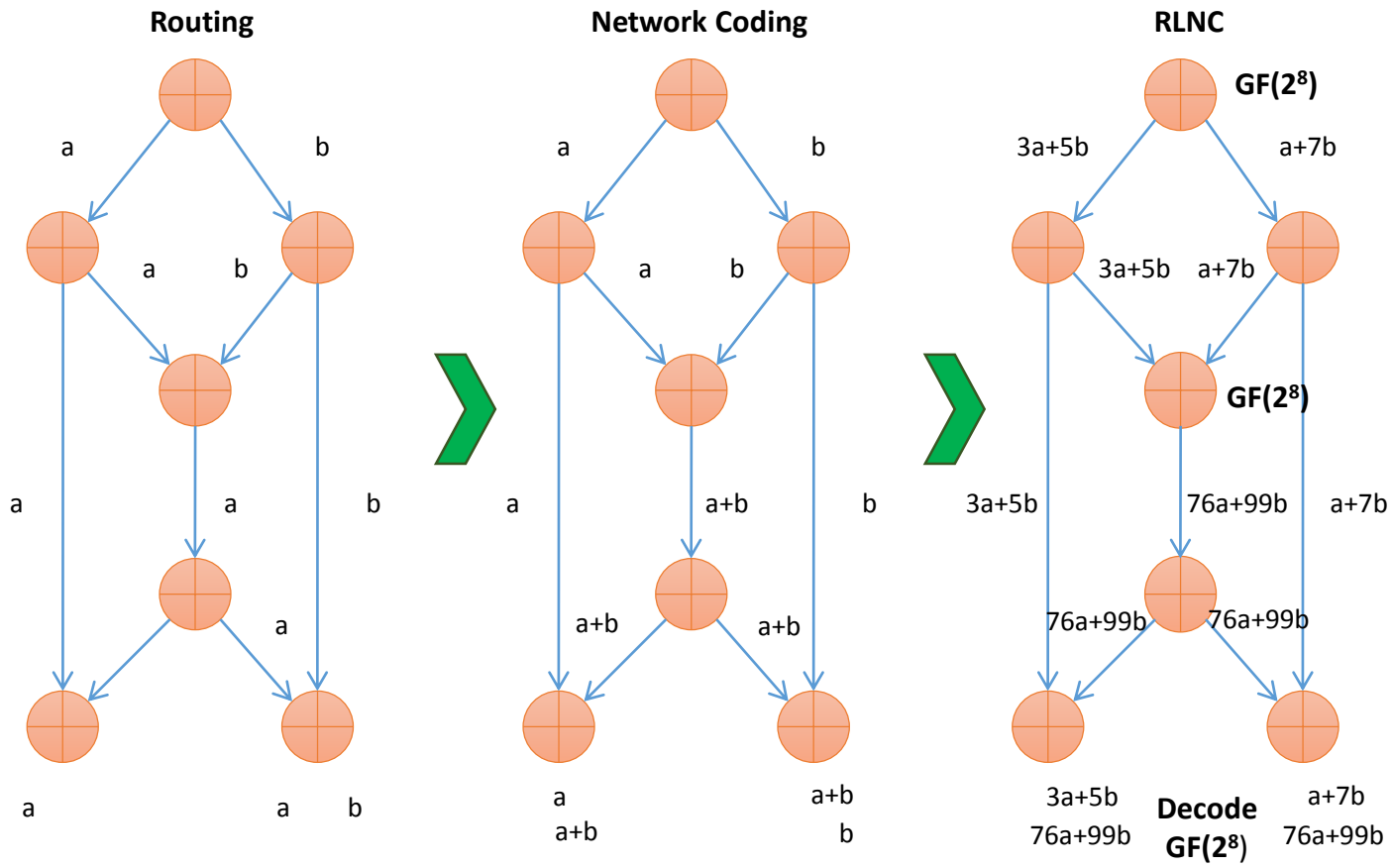
a b | a b

Rate: 2 symbols/time
Centralized, Planned
Optimal
Low-Medium processing cost
One Finite Field in use
Does not consider device capabilities
- One encoder, one decoder
- One recoder

a b | a b

Rate: 2 symbols/time
Distributed (not planned)
Optimal (high probability)
High processing cost
One Finite Field in use
Does not consider device capabilities
- One encoder, one decoder
- One recoder

Network Coding



What is the difference with respect to throughput and signalling?

Rate: 1.5 symbols/time
Distributed (but planned)
Sub-optimal
Low processing cost

Rate: 2 symbols/time
Centralized, Planned
Optimal
Low-Medium processing cost
One Finite Field in use
Does not consider device capabilities
- One encoder, one decoder
- One recoder

Rate: 2 symbols/time
Distributed (not planned)
Optimal (high probability)
High processing cost
One Finite Field in use
Does not consider device capabilities
- One encoder, one decoder
- One recoder

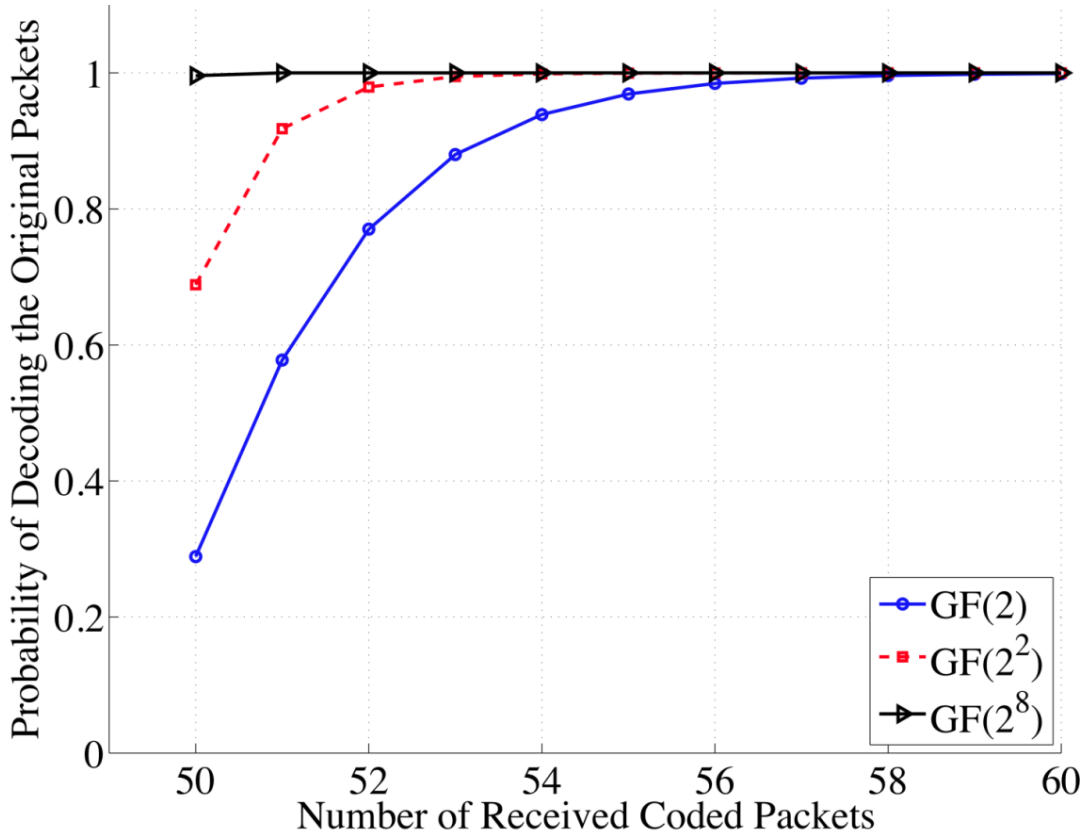
Field Size Analysis

Ralf Kötter: „How bad is binary?“

Field and Generation Size

Following scenario: we have an error-free communication and want to convey 50 packets from A and B.

- Without coding, we would send exactly send 50 packets.
- With Reed-Solomon coding, we would send exactly 50 packets.
- If we used RLNC (without systematic mode), we would need to send more packets, because of linear dependency of the packets (doubles so to speak).



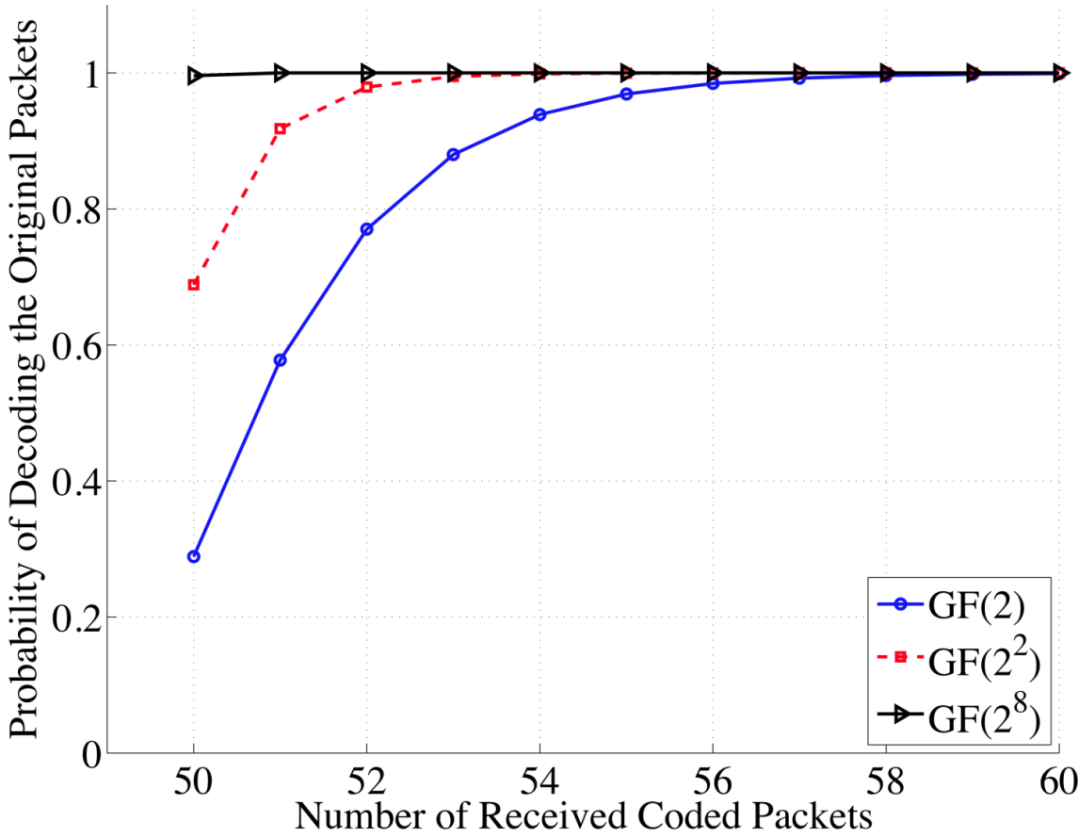
Field and Generation Size

Small field sizes are resulting in linear dependent coded packets.

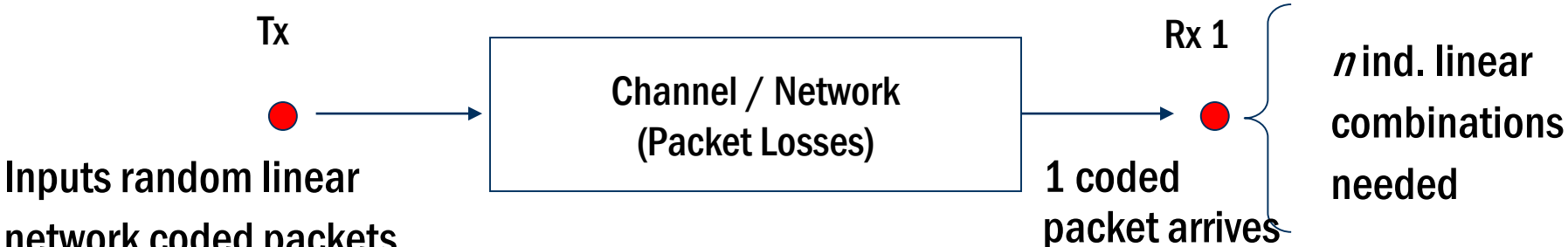
1.6 packets extra per generation in case of binary field sizes.

Large fields sizes (2^8 or higher) have nearly no linear dependency.

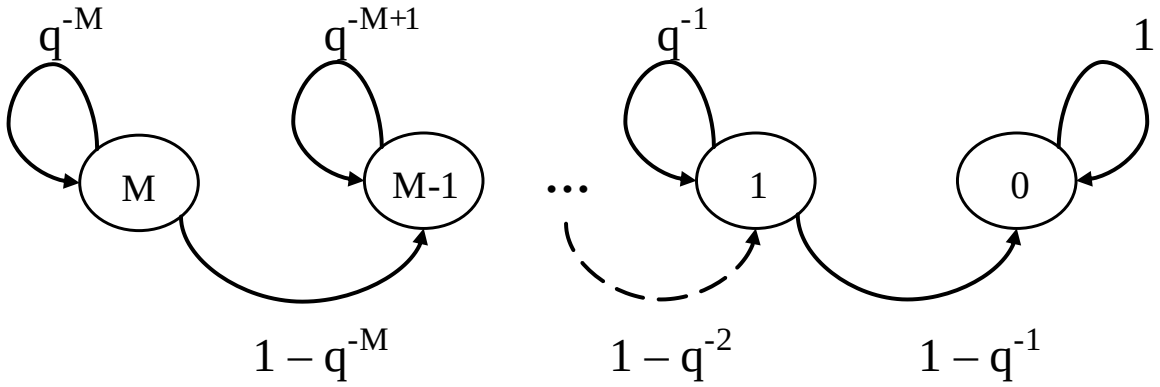
Theory is aiming for large field sizes and large generation sizes!



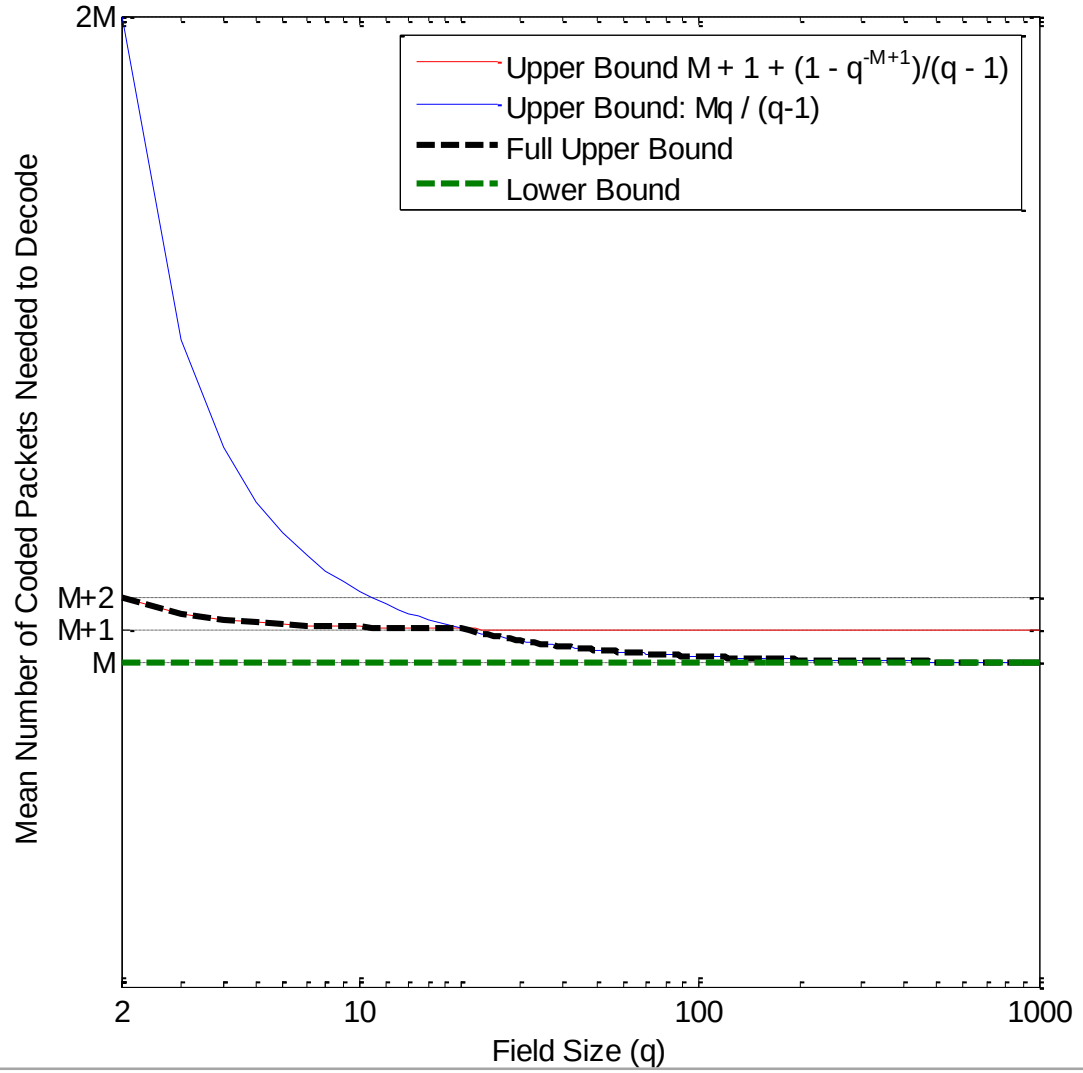
Field Size Analysis for RLNC

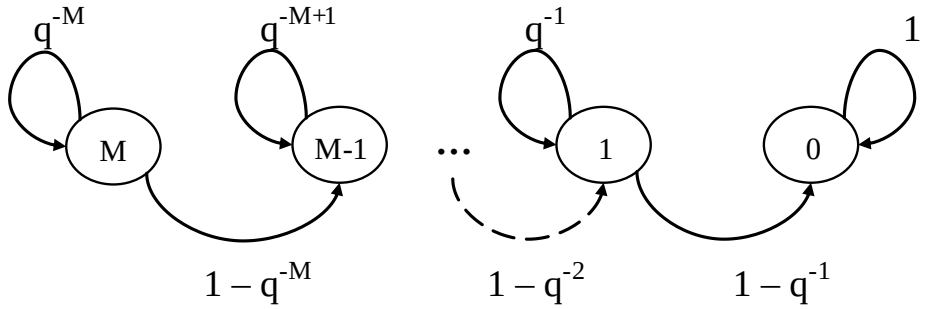


—Modeled as a Markov chain



Field Size Analysis





$$E[N_C] = \sum_{k=1}^{M} \frac{1}{1-q^{-k}}$$
$$\leq min\left(M\frac{q}{q-1}, M+1+\frac{1+q^{-M+1}}{q-1}\right)$$
$$\leq M+2$$

If *M* or q is large:

—Little overhead

—Small performance degradation

Field Size Analysis

GF(2) G=4

0000 → null vector → probability 1 / 2^4 = 1/16

E.g. assuming three linear independent combinations

0110

1000

1101

KODO

1000

0110

0011 ← 0101 (XOR 0110) ← 1101 (XOR 1000)

One more step

1000

0101 (0110 XOR 0011)

0011

What is the probability to achieve a linear independent linear combination?

xxx1

(any combination with a ONE the last digit)

There are eight combinations out of 16 that fulfill that requirement, therefore 50% is the likelihood that we will receive a valuable combination for the last missing packet.

And for the second last combination? Here it is 25%! More in the exercise!

Therefore, the mean value for transmissions over an error-free channel with binary coding is

$$E_n = G + 1.6$$

with G being the generation size.

Field Size Analysis

GF(2) G=4

0000 → null vector → probability 1 / 2^4 = 1/16

E.g. assuming three linear independent combinations

0110 ← SEEN X_2 X_3

1000 ← DECODED X_1

1101 ← SEEN X_1 X_2 X_4

KODO

1000

0110

0011 ← 0101 (XOR 0110) ← 1101 (XOR 1000)

One more step

1000

0101 (0110 XOR 0011)

0011

What is the probability to achieve a linear independent linear combination?

xxx1

(any combination with a ONE the last digit)

There are eight combinations out of 16 that fulfill that requirement, therefore 50% is the likelihood that we will receive a valuable combination for the last missing packet.

And for the second last combination? Here it is 25%! More in the exercise!

Therefore, the mean value for transmissions over an error-free channel with binary coding is

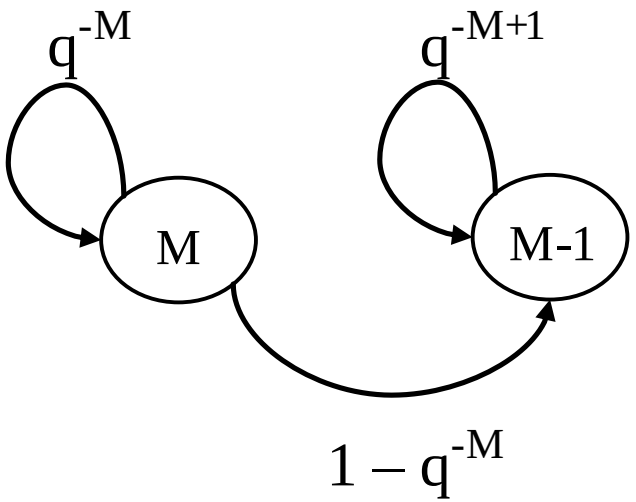
$$E_n = G + 1.6$$

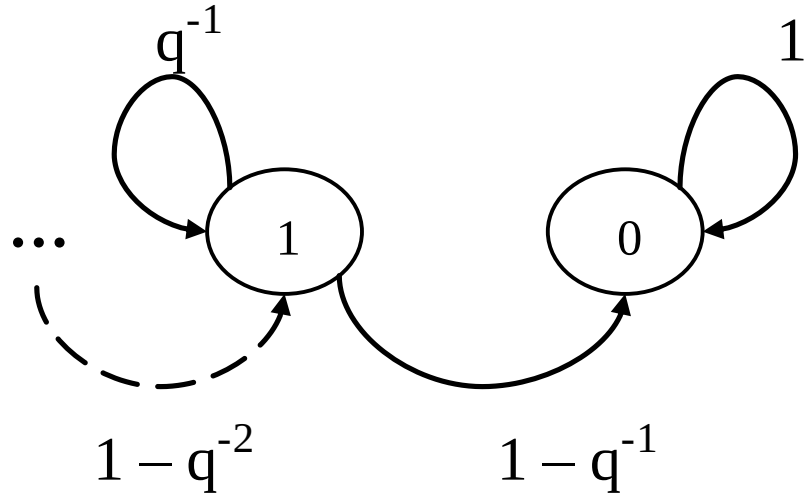
with G being the generation size.

Field Size Analysis

GF(2) G=4

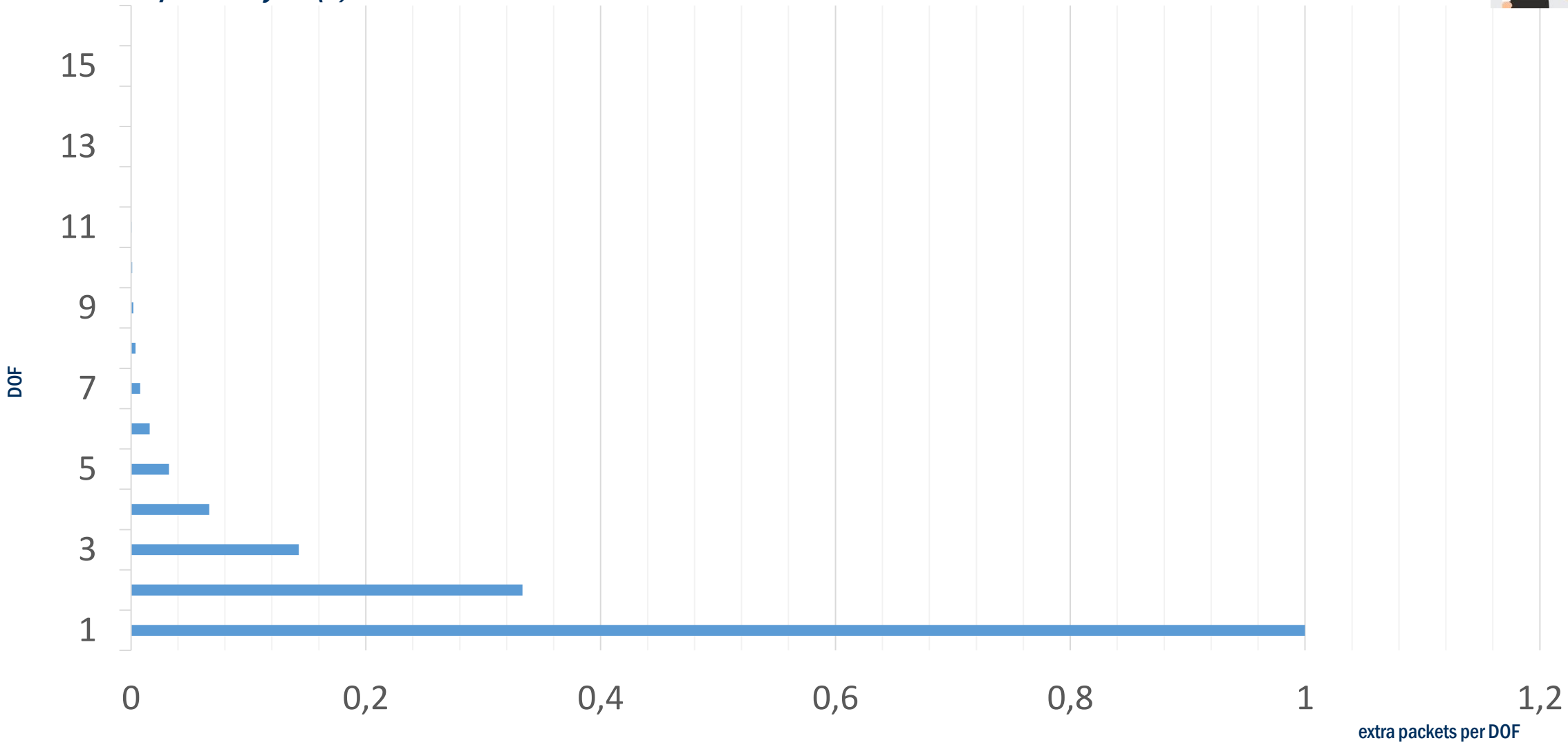
GF(2^8)	$(1/2^8)^4$	$(1/2^8)^3$	$(1/2^8)^2$	$1/2^8$	DONE
GF(2)	1/16	1/8	1/4	1/2	DONE
	0000	x000	xx00	xxx0	





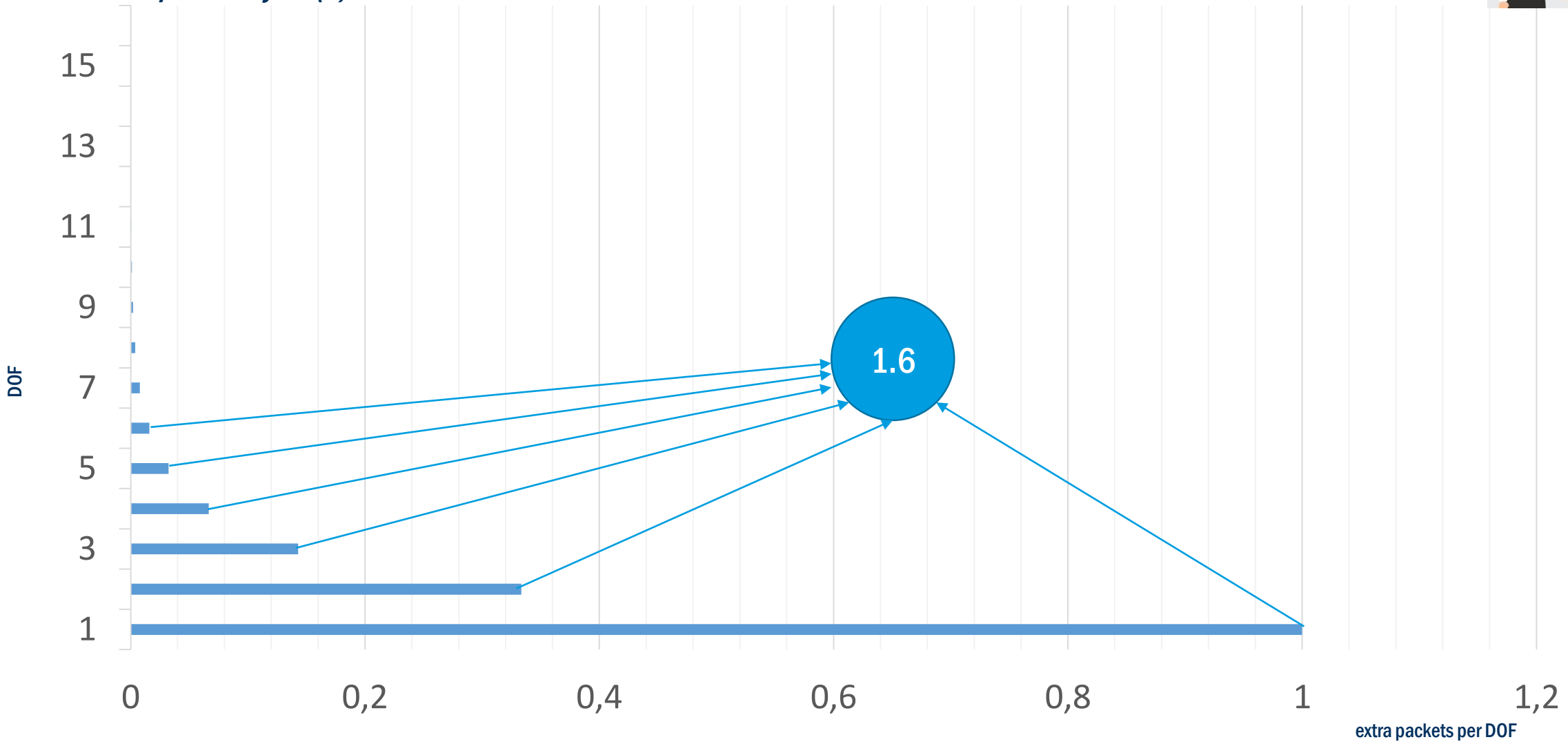
Field Size Analysis

Linear dependency GF(2) G=16



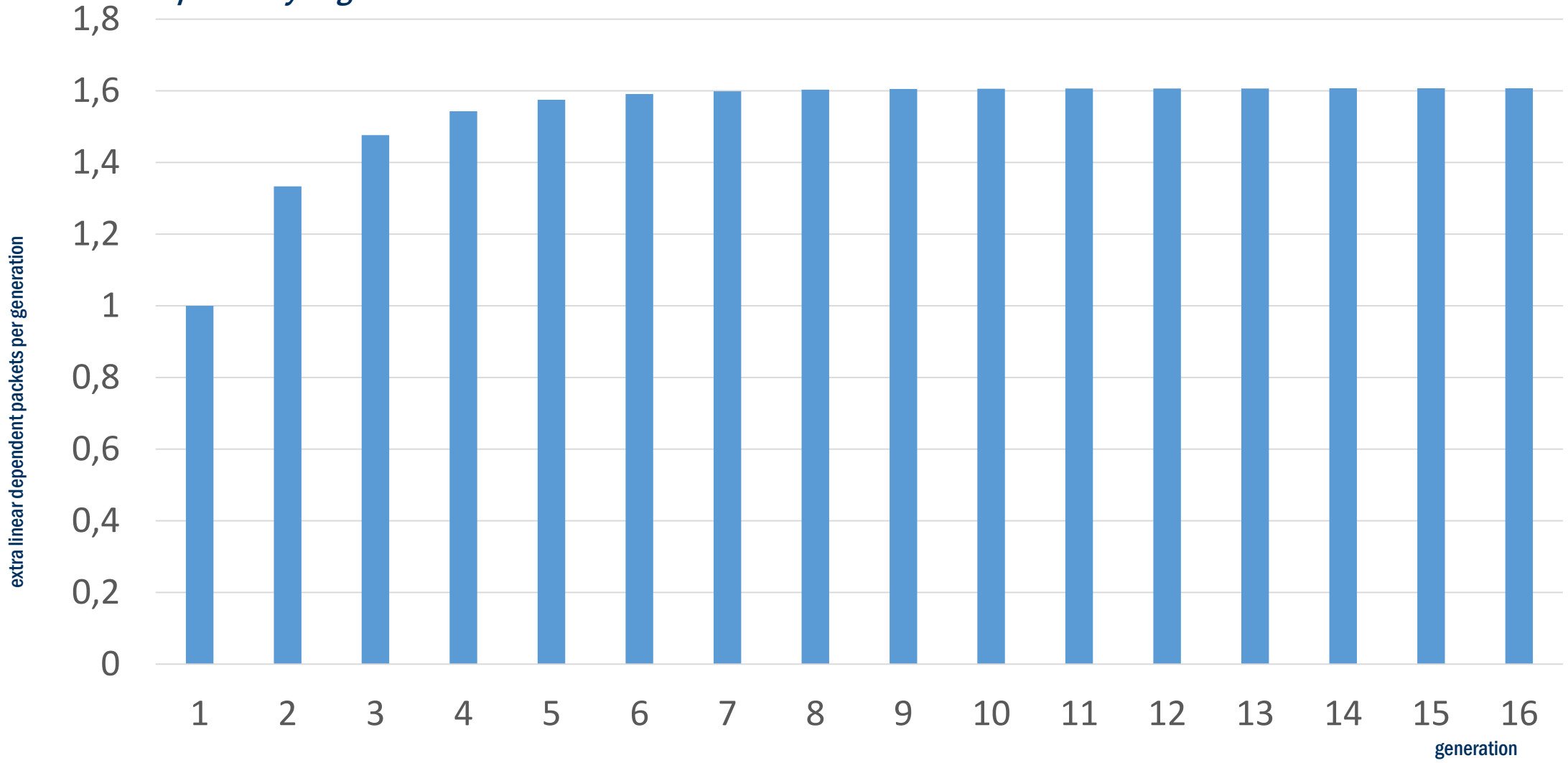
Field Size Analysis

Linear dependency GF(2) G=16



Field Size Analysis

Overhead in dependency of generation size



http://notebook.deutsche-telekom.rocks

In [1]:
```
# Copyright Steinwurf ApS 2015.
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
# See accompanying file LICENSE.rst or
# http://www.steinwurf.com/licensing
```

Basic example: Encoding and Decoding

We need to import kodo. It is simple

In [2]:
```
import os
import sys

import kodo
```

We set some constants

In [3]:
```
# Set the number of symbols (i.e. the generation size in RLNC
# terminology) and the size of a symbol in bytes
symbols = 8
symbol_size = 160
```

To create an encoder and a decoder, we need to create a factory. After, we can build as many encoders and decoders as we want

In [4]:
```
# In the following we will make an encoder/decoder factory.
# The factories are used to build actual encoders/decoders
encoder_factory = kodo.FullVectorEncoderFactoryBinary(symbols, symbol_size)
encoder = encoder_factory.build()

decoder_factory = kodo.FullVectorDecoderFactoryBinary(symbols, symbol_size)
decoder = decoder_factory.build()
```

http://notebook.deutsche-telekom.rocks

Extra packets per generation

How many extra packet do we need to send due to linear dependencies?

In [9]:
```
packets_sent = []
for i in xrange(1000):
    packet_number = 0
    decoder = decoder_factory.build()
    while not decoder.is_complete():
        # Generate an encoded packet
        packet = encoder.write_payload()

        # Pass that packet to the decoder
        decoder.read_payload(packet)
        packet_number += 1

    packets_sent.append(packet_number)

# We print the average of extra sent packets
mean_extra_packets = sum(packets_sent)/float(len(packets_sent)) - symbols
mean_extra_packets
```

Out[9]:
```
1.5899999999999999
```

It is only 1.6 extra packets!

Recoding Potential

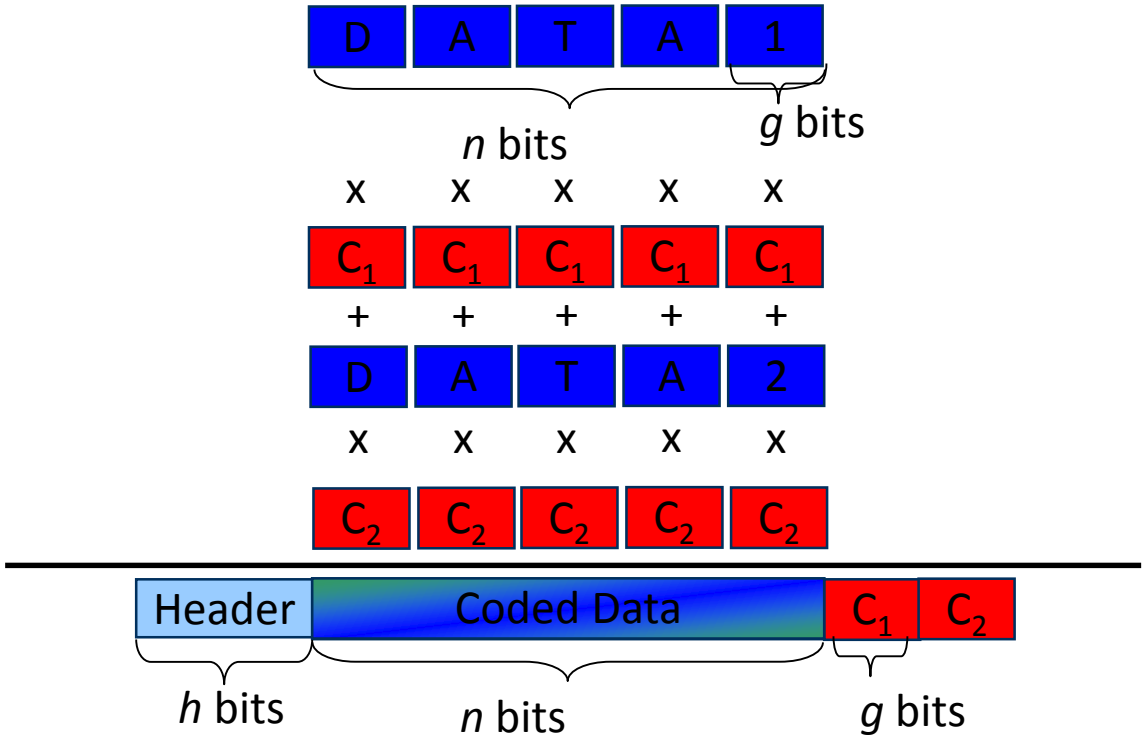
Generating a Coded Packet

- Generating a linear network coded packet (CP)

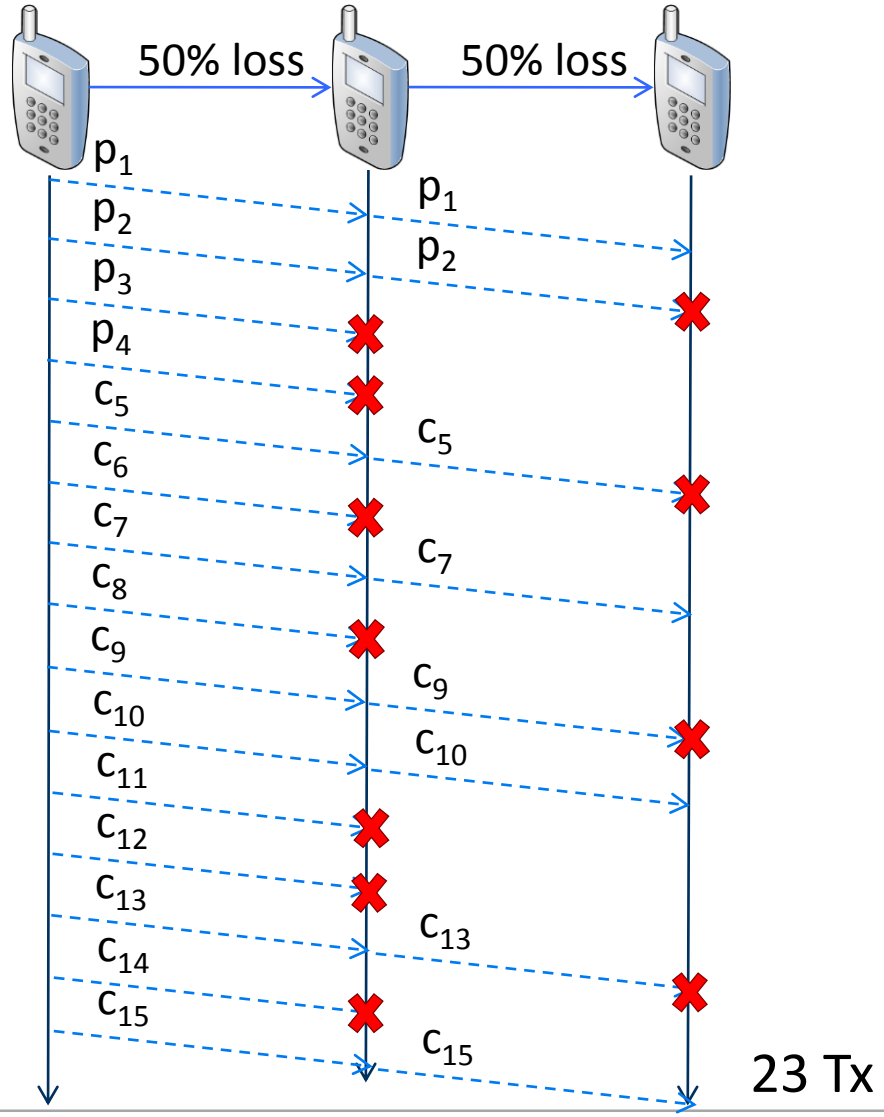
$$CP_j = \sum_i C_i P_i$$

- Operations over finite field of size.

e.g. g= 8 bits, q = 256



No Recoding

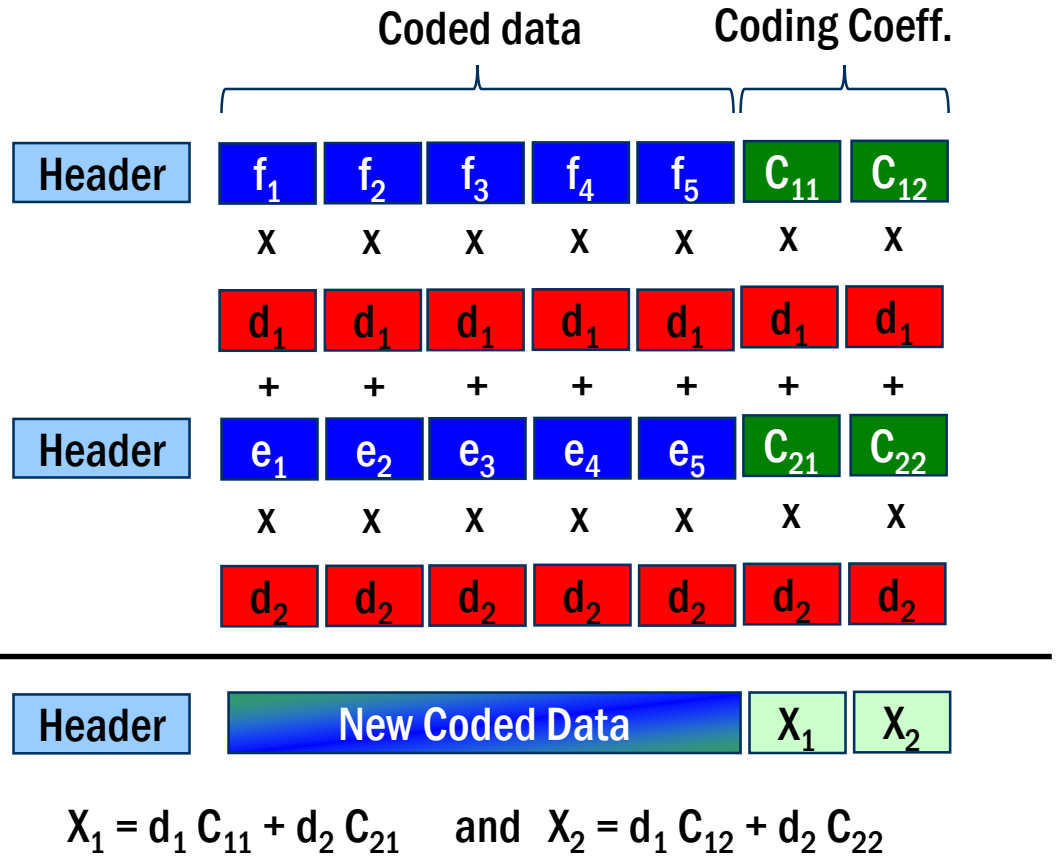


- Simple operation: forward
- Structure of code is preserved
- Issues
  - Delay per batch of packets
  - Missing transmission opportunities
  - Equivalent loss probability: compounding each channel's loss

- Loss of channel *i*: e_i
- Equivalent success prob of a packet:
- $(1-e_1)\,(1-e_2)$

Recoding Packets

- Generating a new linear network coded packet (CP)

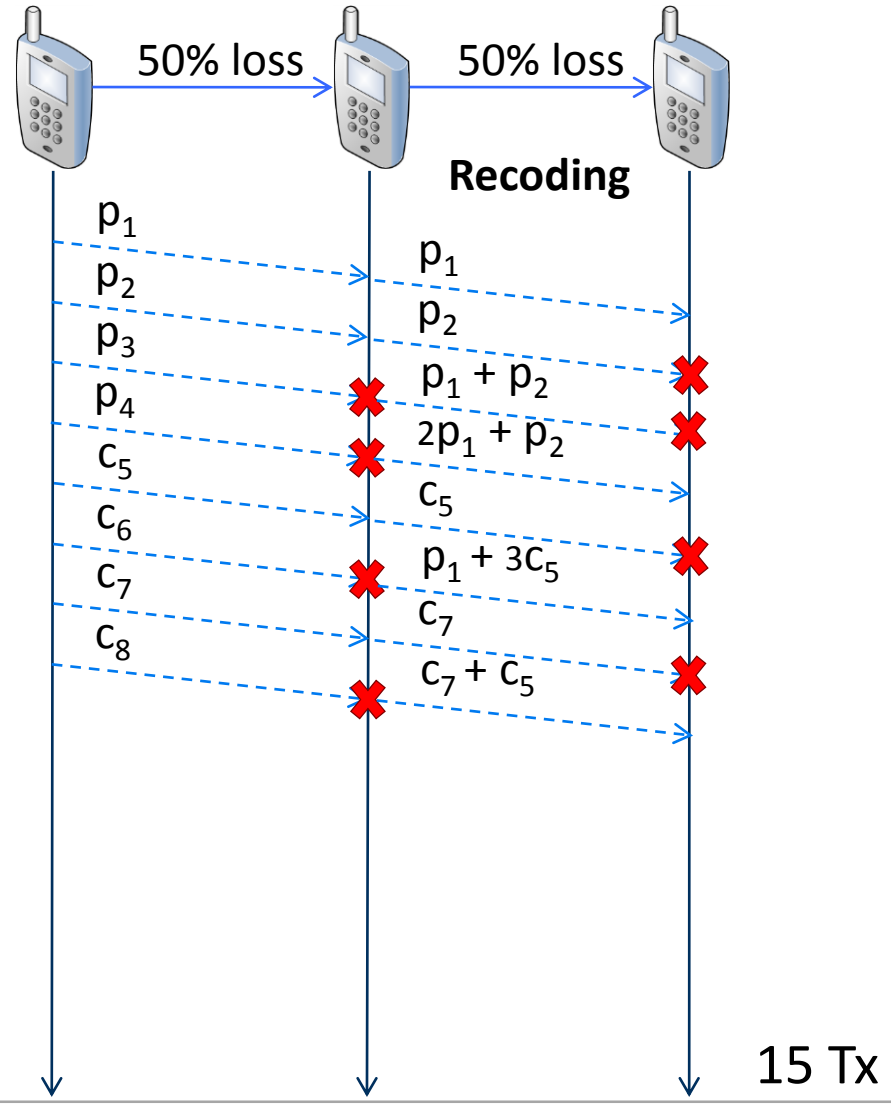


$X_1 = d_1 C_{11} + d_2 C_{21}$ and $X_2 = d_1 C_{12} + d_2 C_{22}$

Recall that: $f = C_{11}P_1 + C_{12} P_2$ and $e = C_{21}P_1 + C_{22} P_2$

Thus, $d_1f + d_2e = d_1C_{11}P_1 + d_1C_{12} P_2 + d_2C_{21}P_1 + d_2C_{22} P_2 = X_1P_1 + X_2P_2$

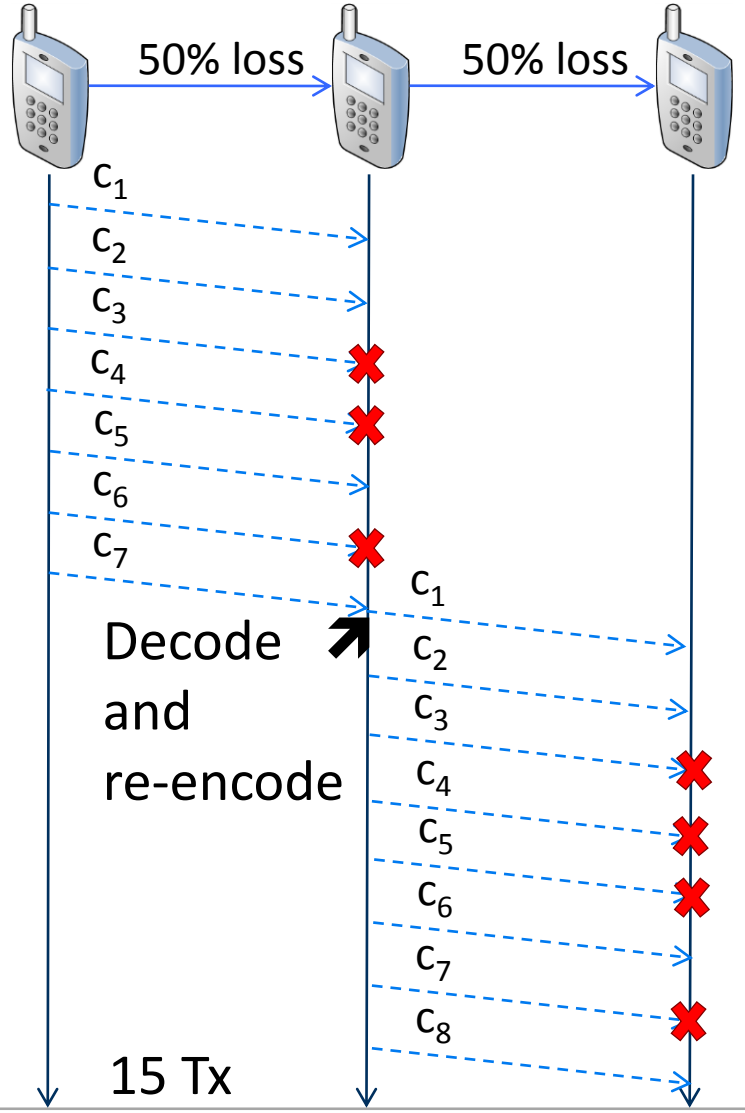
Recoding Packets



- A bit more complex: recode
  - Equivalent to encoding in worst case
  - Not so bad
- Structure of code may be changed
  - Some exceptions
- Issues
  - If left unchecked, can have unnecessary transmissions, e.g., c_8
  - Additional processing needed
- Advantage: equivalent success probability of a *linear combination*:

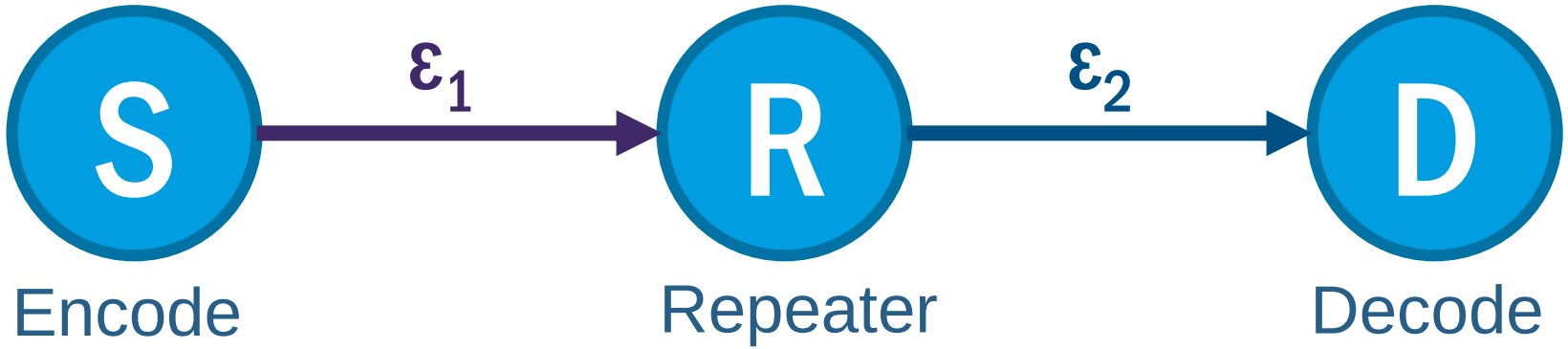
  $\min\{1-e_1, 1-e_2\}$

Is “recoding” possible with other linear codes?



- Consider Reed-Solomon, LT, etc
- Structure is not composable
  - Mixing coded packets does not
  - produce a “valid” coded packet
  - Different structure, properties are lost
- Recoding means receiving enough
- coded packets, decode,
- and *then* re-encode
- Issues
  - Delay per batch
  - Computational effort
  - Can also have unnecessary Tx
- Success probability of a *linear combination*: $\min\{1-e_1, 1-e_2\}$

Network Coding

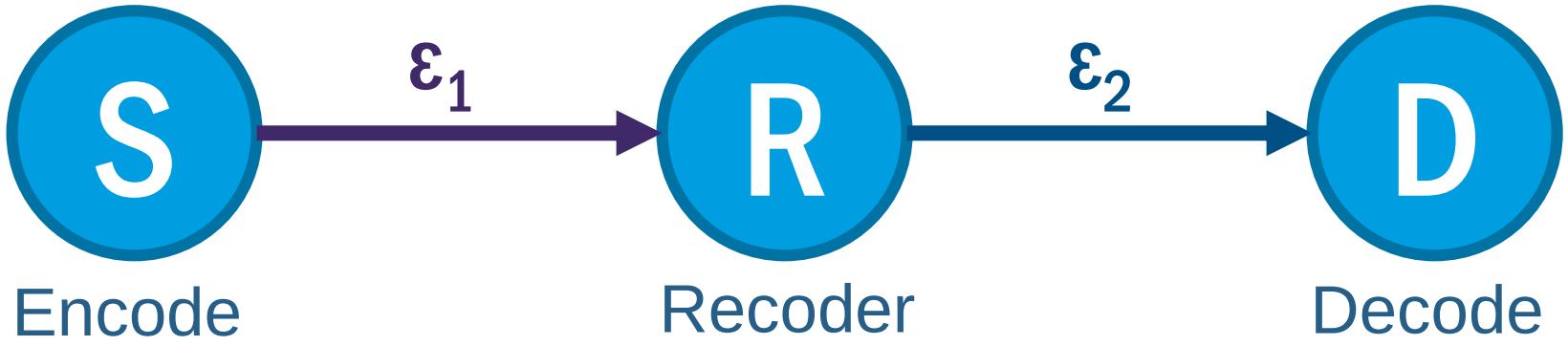


Reed Solomon
LDPC
LT
Raptor

$(1-\varepsilon_1)(1-\varepsilon_2)$

Store and Forward

Network Coding



Reed Solomon
LDPC
LT
Raptor

$(1-\varepsilon_1)(1-\varepsilon_2)$

Store and Forward

Network Coding

$\min\{(1-\varepsilon_1);(1-\varepsilon_2)\}$

Compute and Forward

http://notebook.deutsche-telekom.rocks

- encoding-decoding.ipynb example

- encoding-recoding-decoding.ipynb example
- Proof the aforementioned gain of recoder over repeater
  - Losses of link one and two are 60% and 20%, respectively
- The „shark fin“ problem
- https://arxiv.org/abs/1601.03201

http://notebook.deutsche-telekom.rocks

In [1]:
```
# Copyright Steinwurf ApS 2015.
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
# See accompanying file LICENSE.rst or
# http://www.steinwurf.com/licensing

import os
import sys
import random

import kodo

"""
Encode recode decode example.
In Network Coding applications one of the key features is the
ability of intermediate nodes in the network to recode packets
as they traverse them. In Kodo it is possible to recode packets
in decoders which provide the recode() function.
This example shows how to use one encoder and two decoders to
simulate a simple relay network as shown below (for simplicity
we have error free links, i.e. no data packets are lost when being
sent from encoder to decoder1 and decoder1 to decoder2):
        +-----------+     +-----------+     +-----------+
        |  encoder  |+---.| decoder1  |+---.|  decoder2 |
        +-----------+     | (recoder) |     +-----------+
                          +-----------+
In a practical application recoding can be using in several different
ways and one must consider several different factors e.g. such as
reducing linear dependency by coordinating several recoding nodes
in the network.
Suggestions for dealing with such issues can be found in current
research literature (e.g. MORE: A Network Coding Approach to
Opportunistic Routing).
"""
```

http://notebook.deutsche-telekom.rocks

Simple forwarding

In [2]:
```
e1 = ['loss'] * 2 + ['success'] * 8
e2 = ['loss'] * 6 + ['success'] * 4

packets_sent = []
for i in xrange(1000):
    decoder1 = decoder_factory.build()
    decoder2 = decoder_factory.build()
    packet_number = 0
    while not decoder2.is_complete():

        # Encode a packet into the payload buffer
        packet = encoder.write_payload()
        packet_number += 1

        # Pass that packet to decoder1 with (1-e1) probability
        if random.choice(e1) == 'success':
            decoder1.read_payload(packet)
        else:
            continue

        # Now produce a new recoded packet from the current
        # decoding buffer, and place it into the payload buffer
        packet = decoder1.write_payload()

        # Pass the recoded packet to decoder2 with (1-e2) probability
        if random.choice(e2) == 'success':
            decoder2.read_payload(packet)

    packets_sent.append(packet_number)

# We print the average packets sent
sum(packets_sent)/float(len(packets_sent))
```

Out[2]: 100.324

http://notebook.deutsche-telekom.rocks

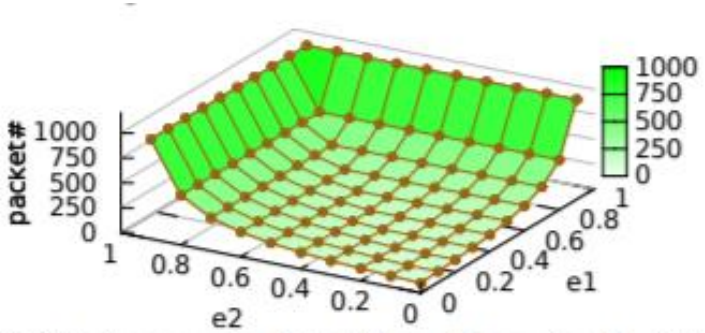
With recoding

In [4]:
```
e1 = ['loss'] * 2 + ['success'] * 8
e2 = ['loss'] * 6 + ['success'] * 4

packets_sent = []
for i in xrange(1000):
    decoder1 = decoder_factory.build()
    decoder2 = decoder_factory.build()
    packet_number = 0
    while not decoder2.is_complete():

        # Encode a packet into the payload buffer
        packet = encoder.write_payload()
        packet_number += 1

        # Pass that packet to decoder1 with (1-e1) probability
        if random.choice(e1) == 'success':
            decoder1.read_payload(packet)

        # Now produce a new recoded packet from the current
        # decoding buffer, and place it into the payload buffer
        packet = decoder1.write_payload()

        # Pass the recoded packet to decoder2 with (1-e2) probability
        if random.choice(e2) == 'success':
            decoder2.read_payload(packet)

    packets_sent.append(packet_number)

# We print the average packets sent
sum(packets_sent)/float(len(packets_sent))
```

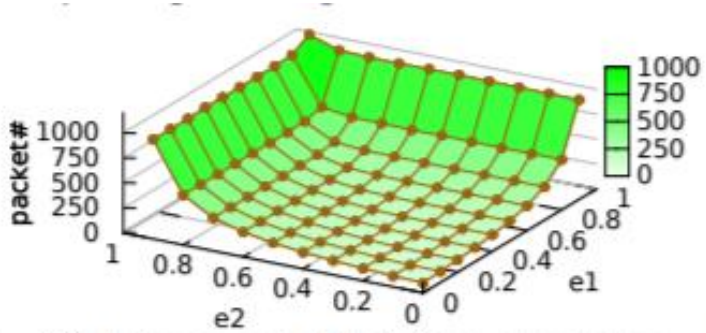
Out[4]: 80.323

Sharkfin Problem

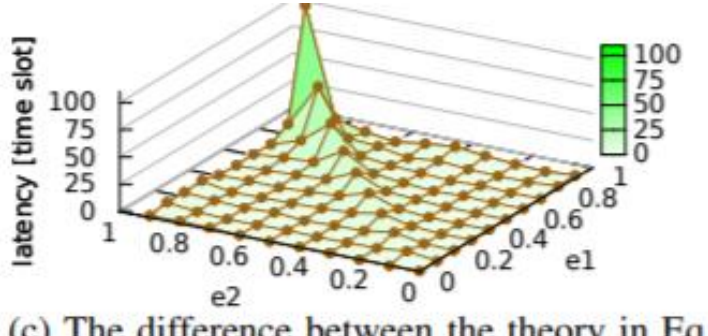
- If error rate is high and the same (or close) for both links, the theory and the practical implementation do not match.
- Underlines the importance for implementation and theory → Theory that matters!



(a) Latency calculated from theory in Eq. (5).



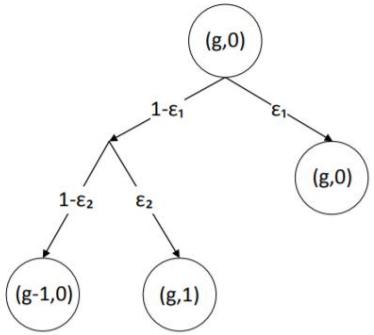
(b) Latency calculated from simulations.



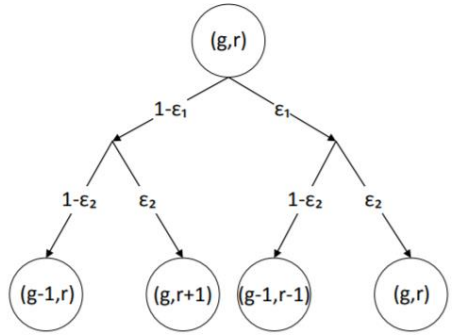
(c) The difference between the theory in Eq. (5) and the simulations.

Sharkfin Problem

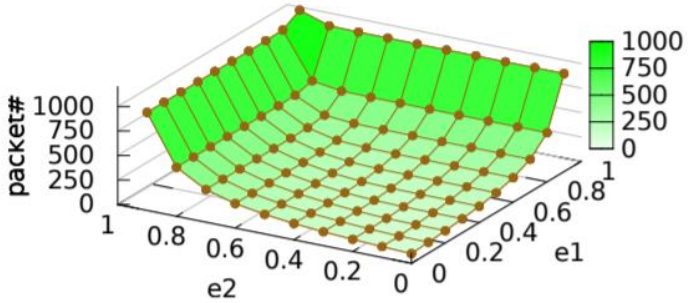
- Adjustment of the theory presented in https://arxiv.org/abs/1601.03201



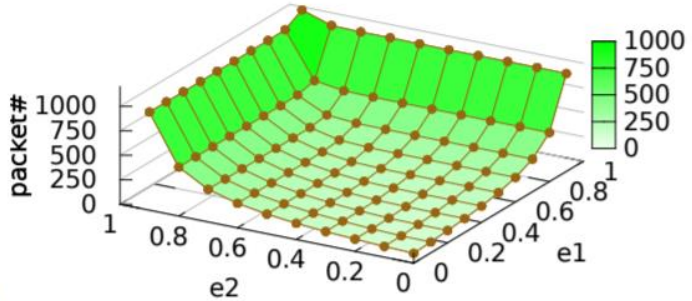
(a) State transition graph when the recoder is empty.



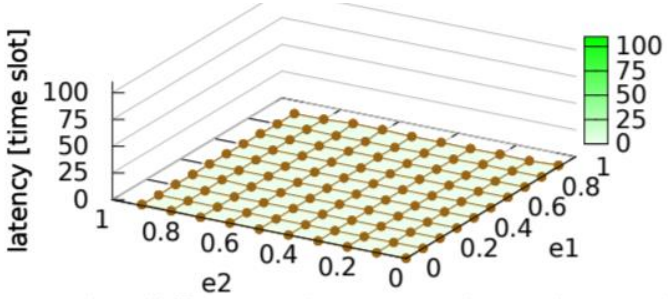
(b) State transition graph when the recoder has at least on linearly independent packet.



(a) Latency calculated from recursive formula in Eq. (7).



(b) Latency calculated from simulations.

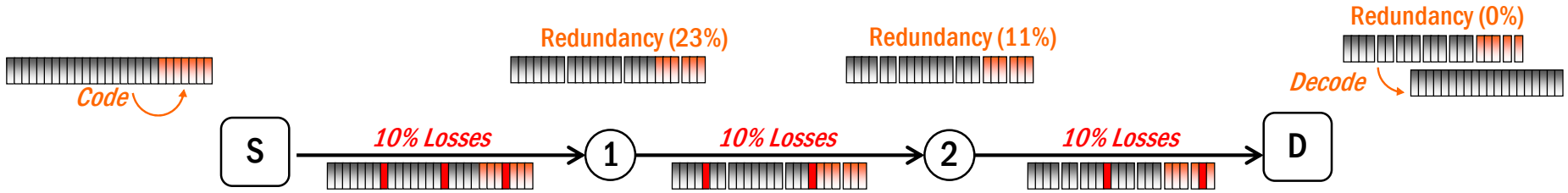


(c) The difference between the values calculated from the recursive formula in Eq. (7) and the simulations.

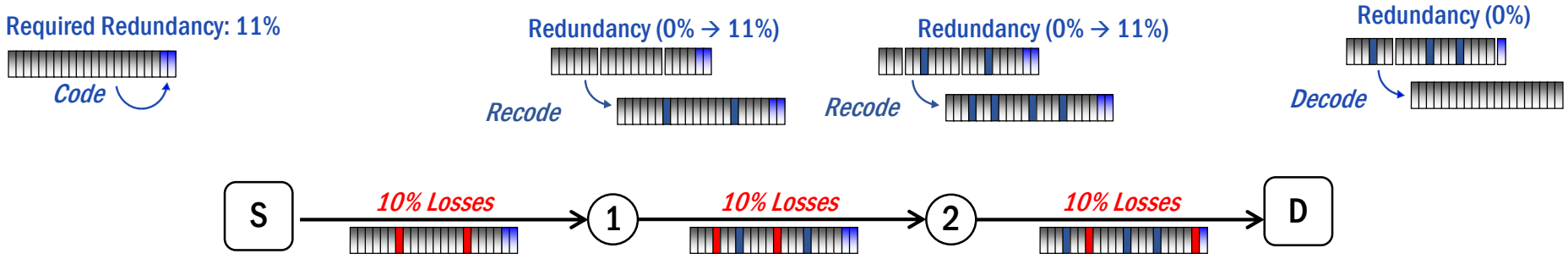
The Recoding Advantage

Optimal and Dynamic Loss Compensation

Coding End-to-End Overhead = Cumulative Losses (37%)



Re-Coding Overhead = Single Worst-Case Loss (11%)



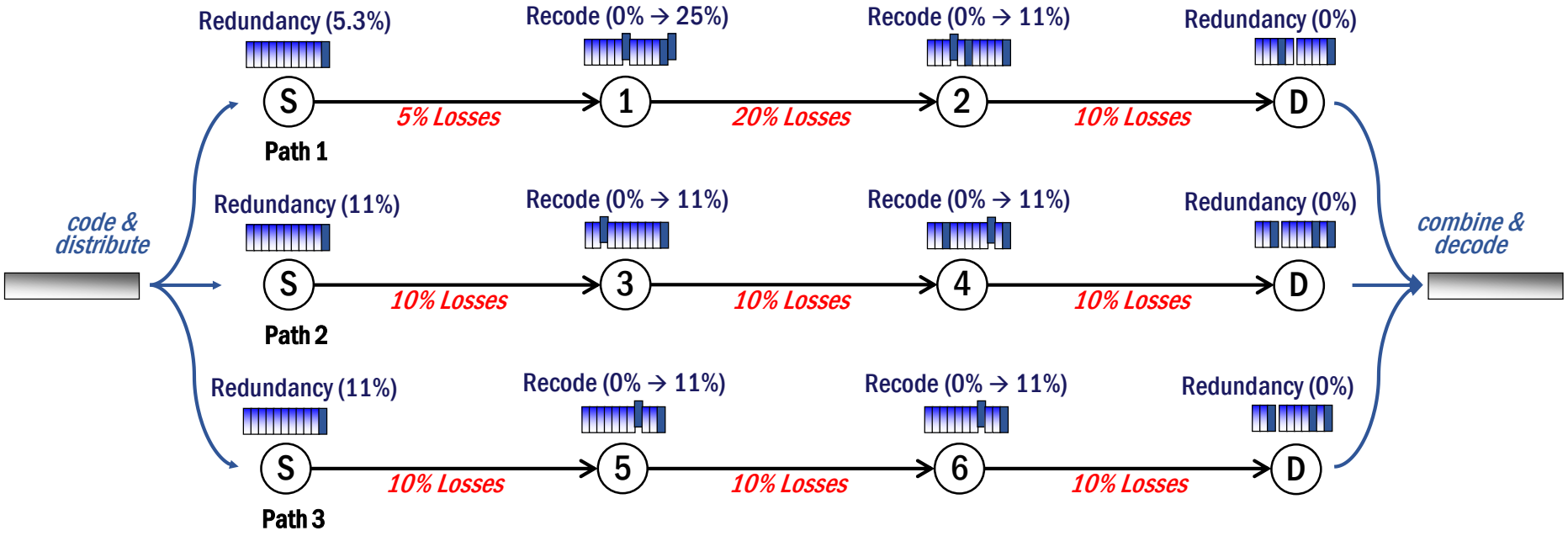
http://notebook.deutsche-telekom.rocks

- Kodo_Multihop_Example.ipynb example
- Gain=#timeslots_of_repeater / #timeslots_of_recoder

#Relays	1	2	3	4	5	6	7	8	9
Repeater									
Recoder									
Gain									

The Recoding Advantage plus Multipath Advantage

Native Bandwidth Aggregation

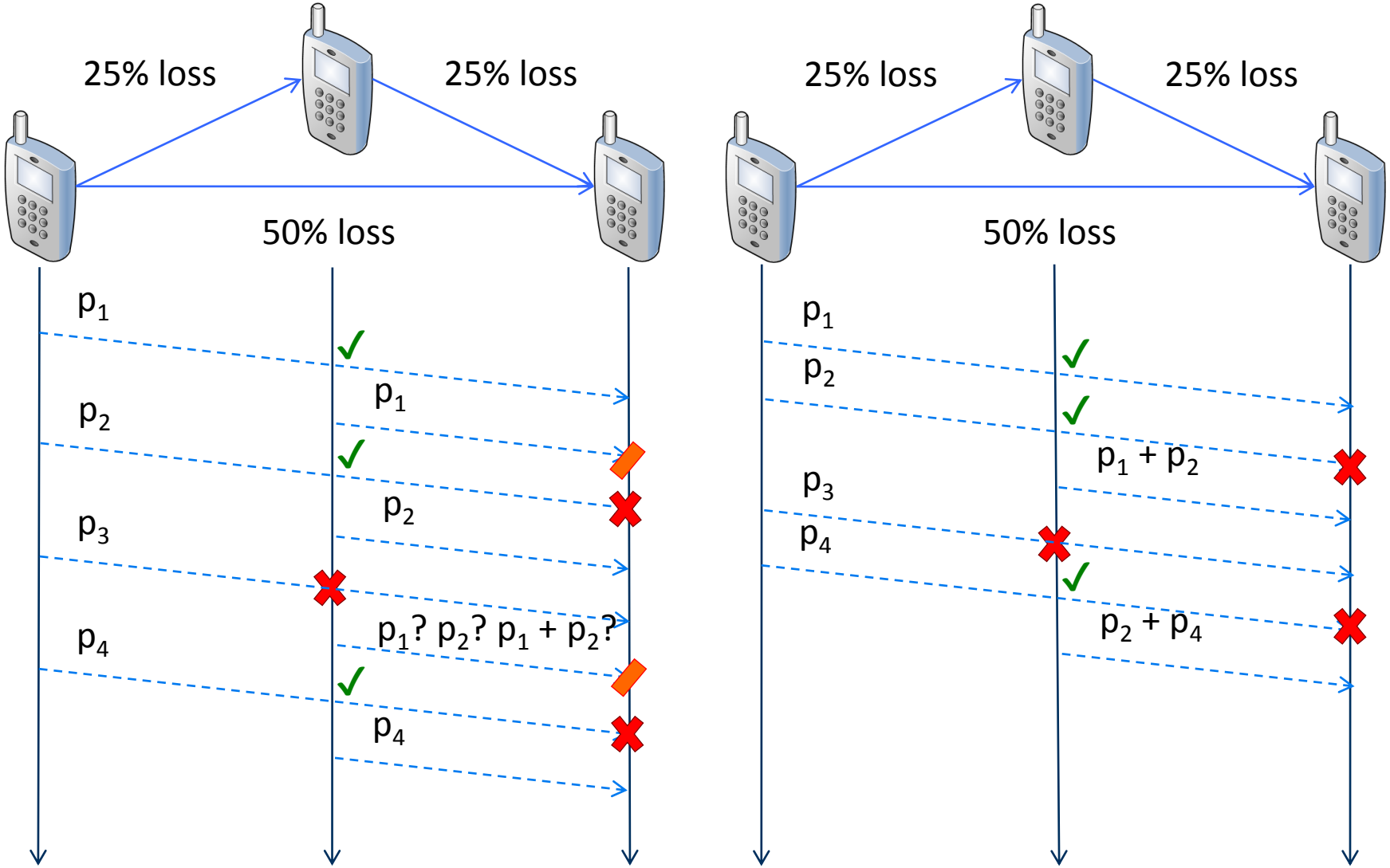


Summing without scheduling

multipath

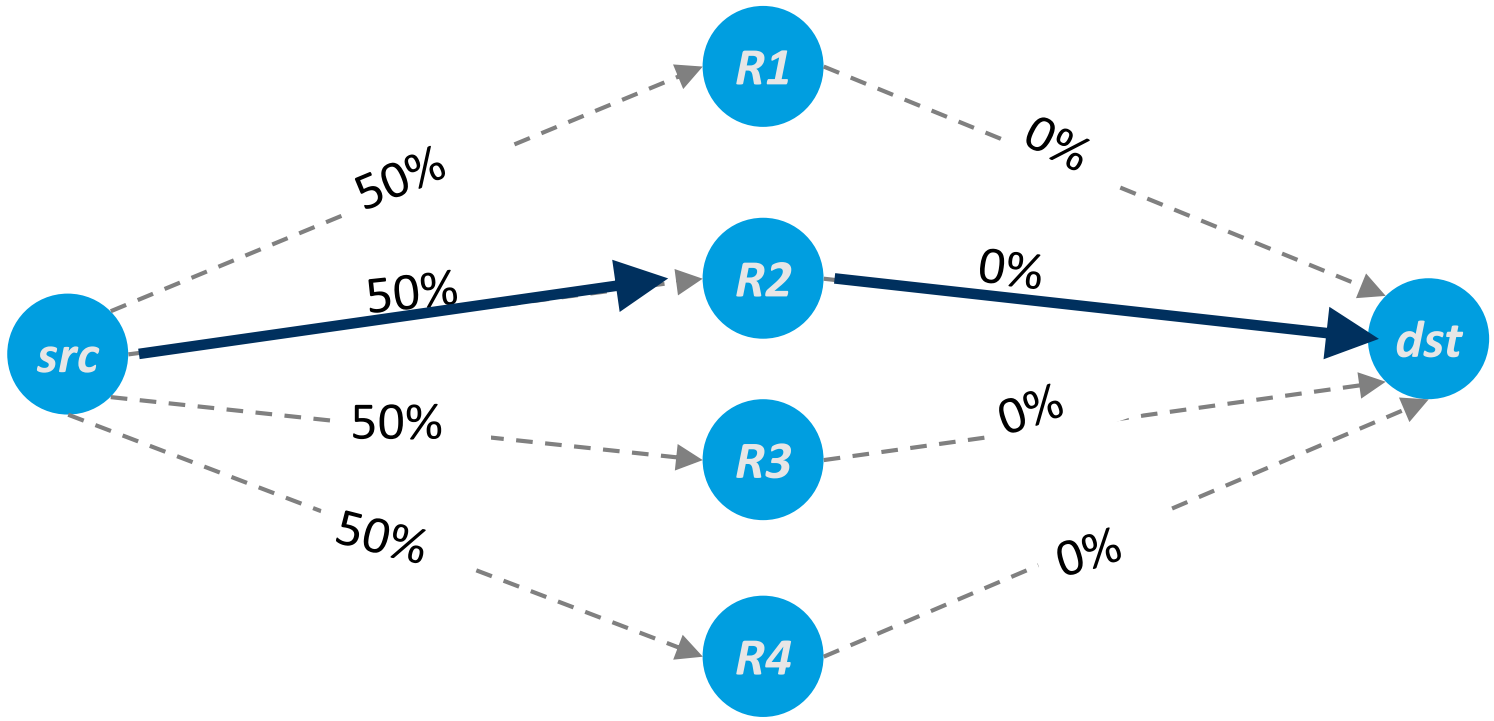
- Total Throughput = $\sum$ (path rate – worst link)
- Inject 10Mpbs at each path → Obtain 26Mbps seamlessly

Other Recoding Issues



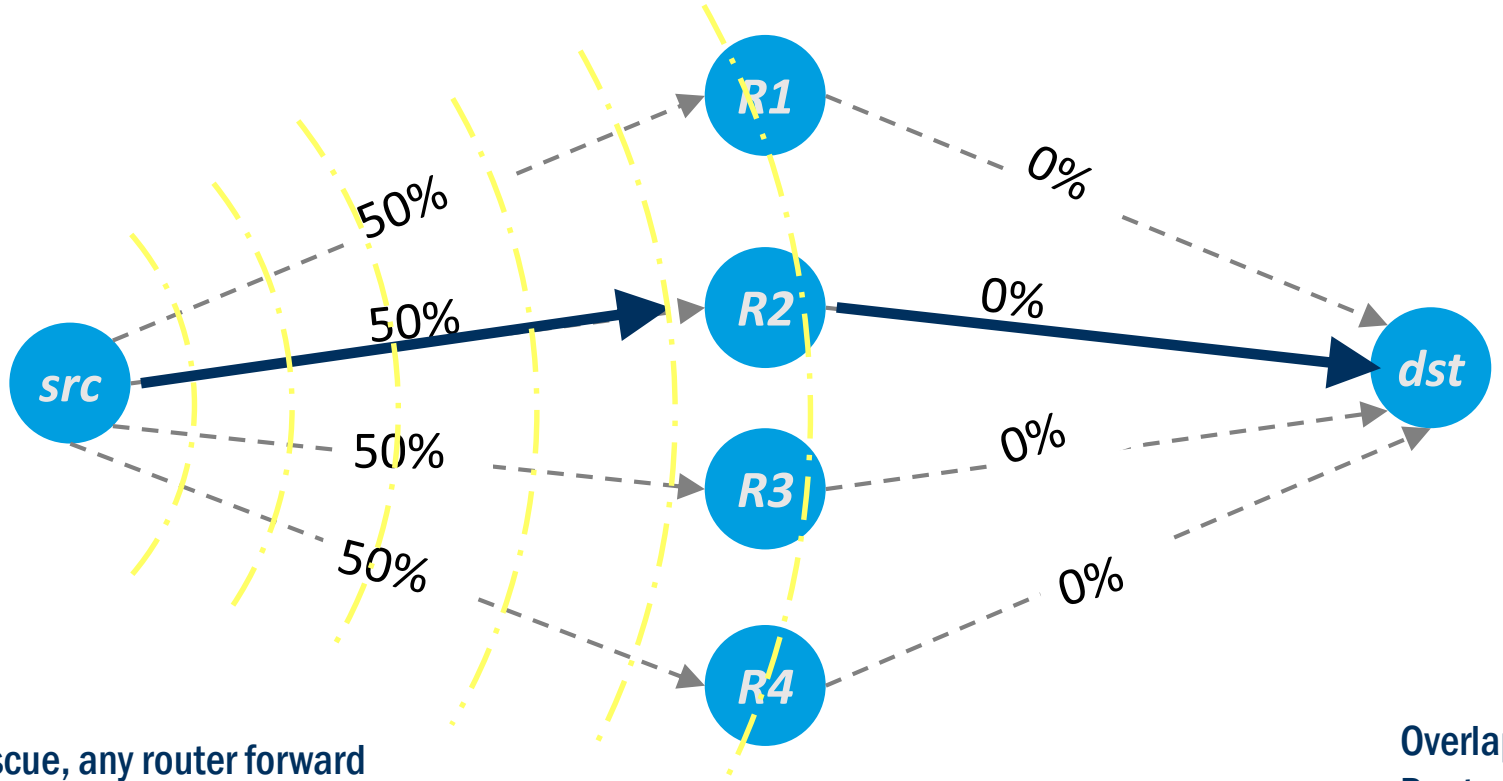
Other Recoding Issues

Best single path → Prob. of loss 50%



Other Recoding Issues

Best single path → Prob. of loss 50%

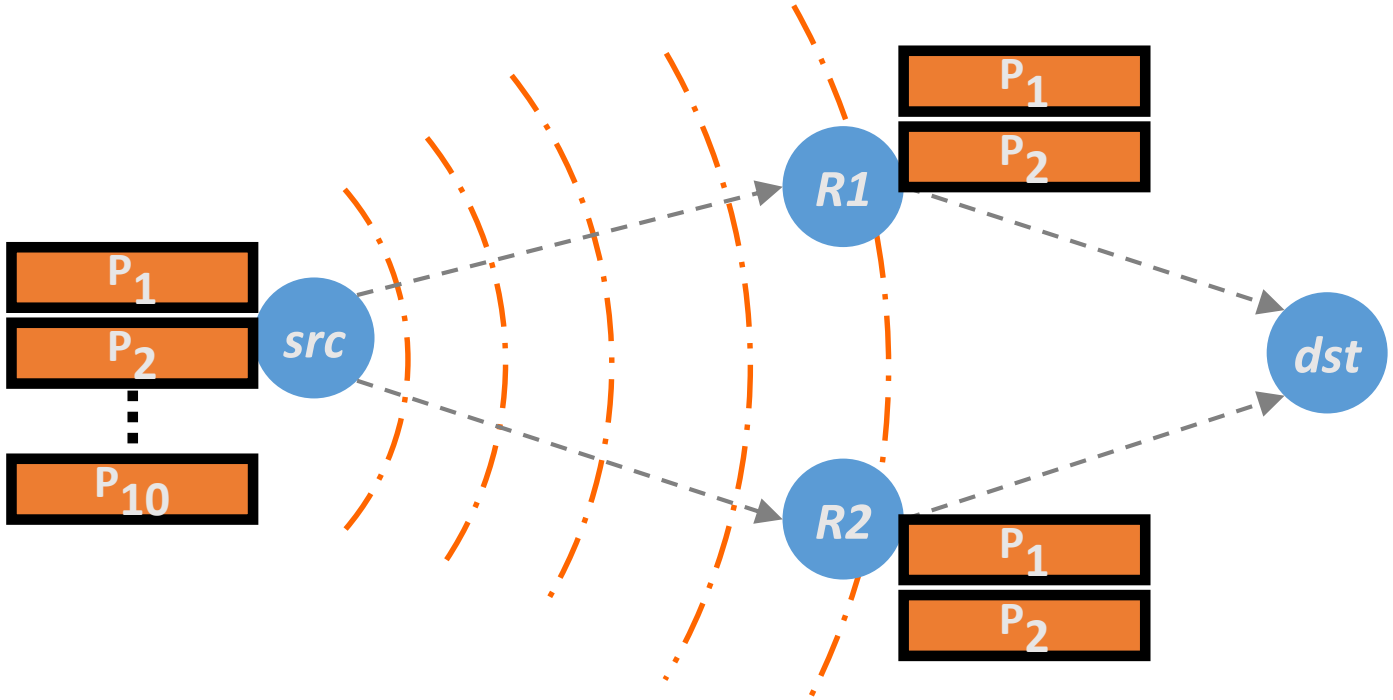


Spatial diversity to the rescue, any router forward packet → Prob. of loss 0.54 = 6%

Overlap in received packets → Routers forward duplicates

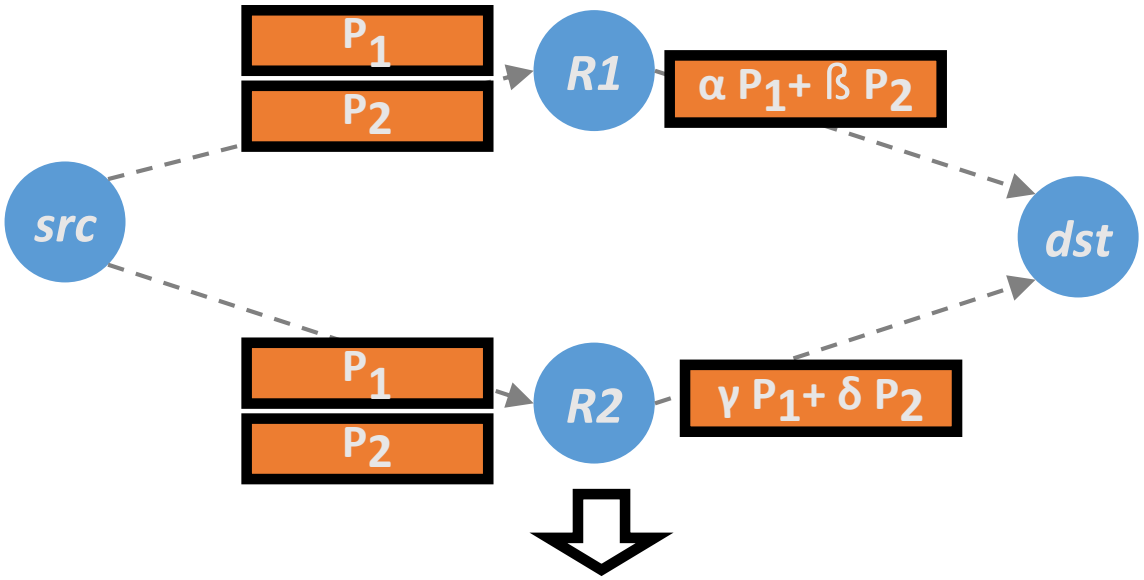
Challenge with Using Spatial Diversity

Overlap in received packets → Routers forward duplicates



Challenge with Using Spatial Diversity

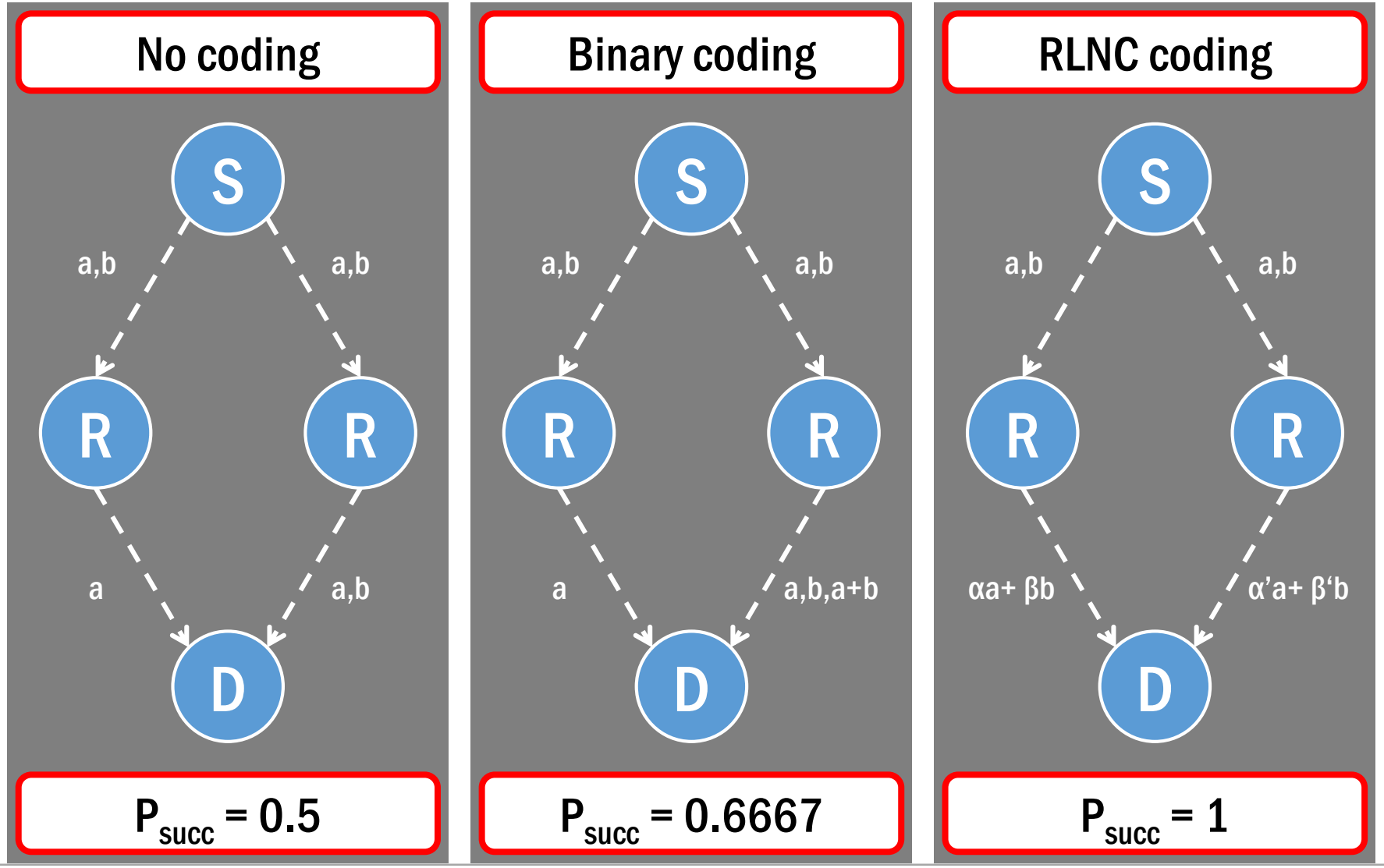
Each router forwards random combinations of packets



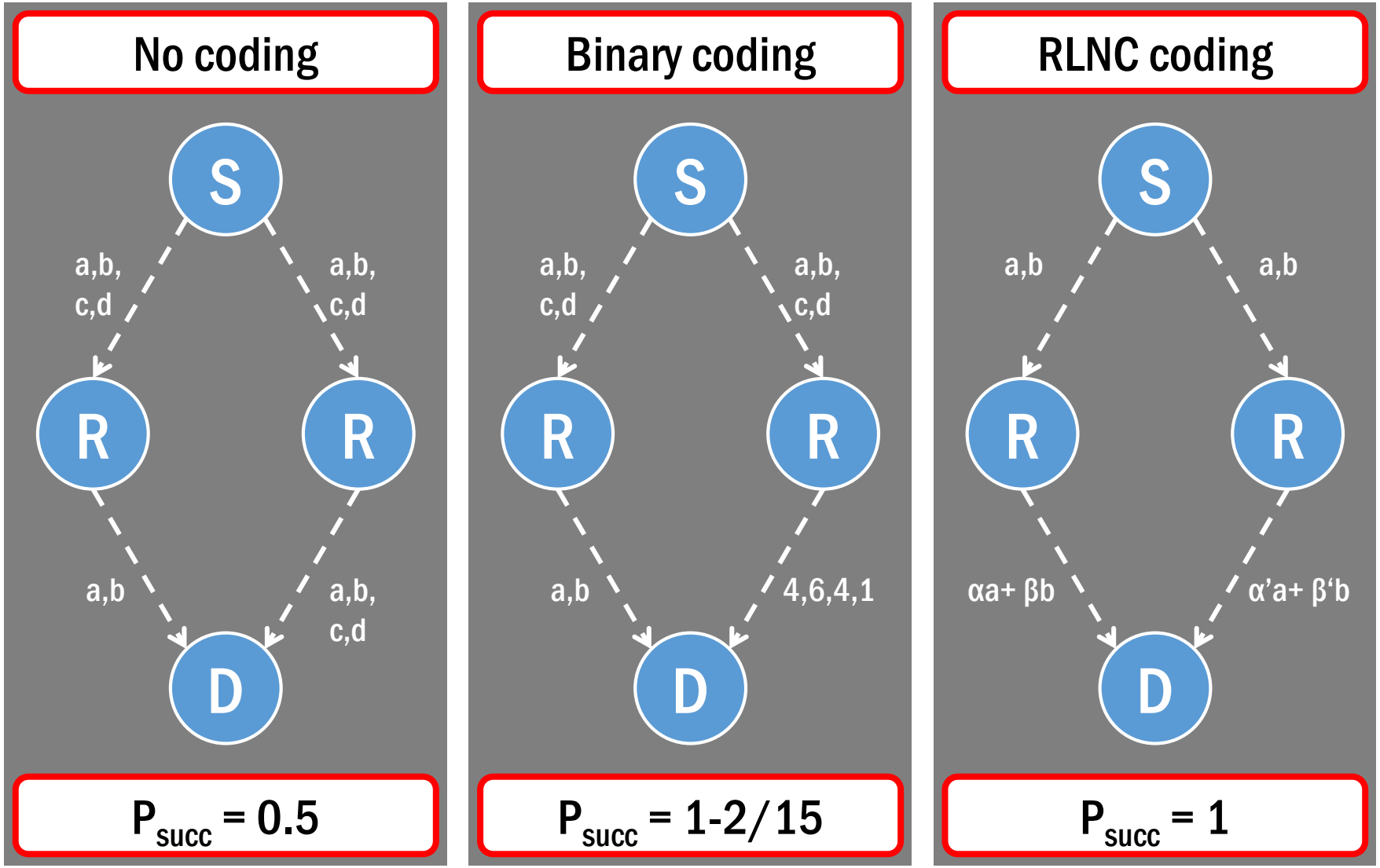
Randomness prevents duplicates

Network coding exploits spatial diversity to improve dead spots

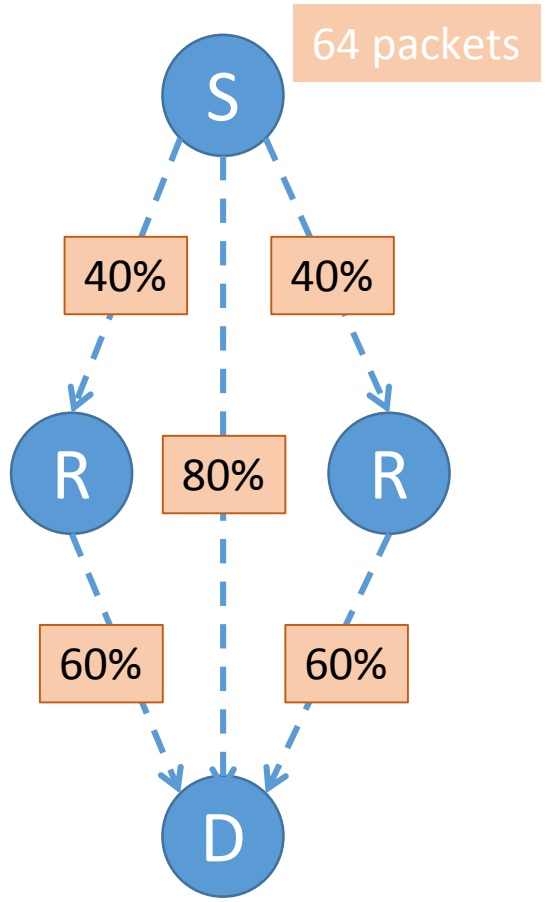
Impact of Recoding



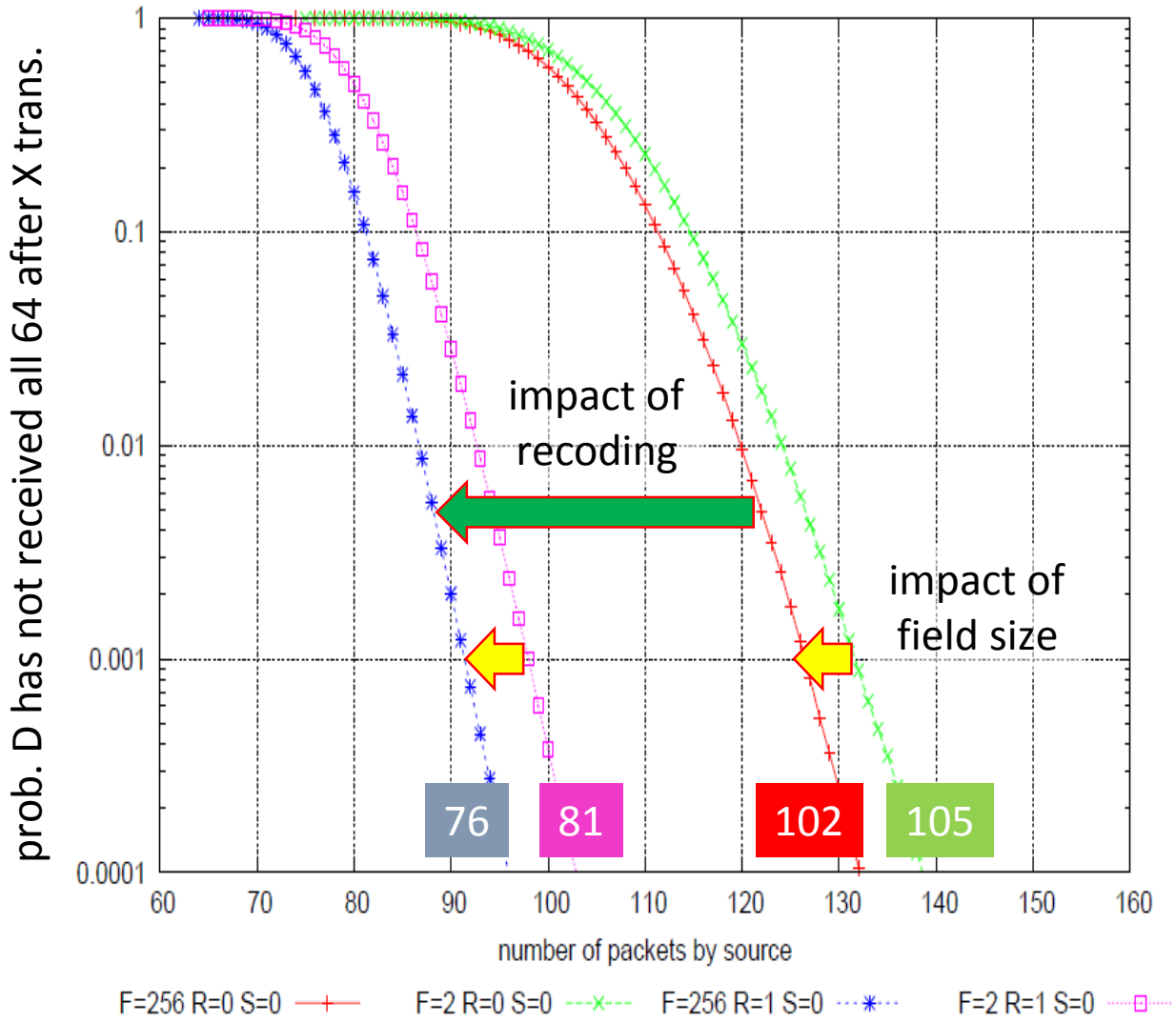
Impact of Recoding



Impact of Recoding

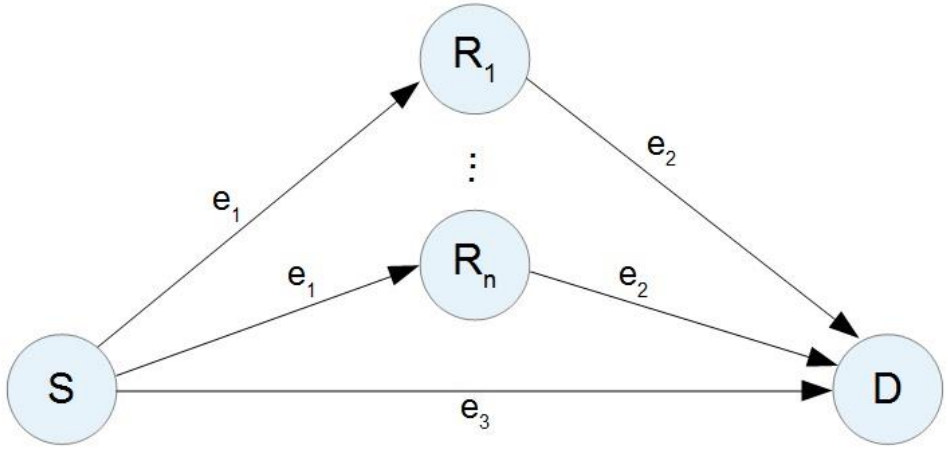


No need for signalling!



http://notebook.deutsche-telekom.rocks

- Kodo_Multipath_Example.ipynb example

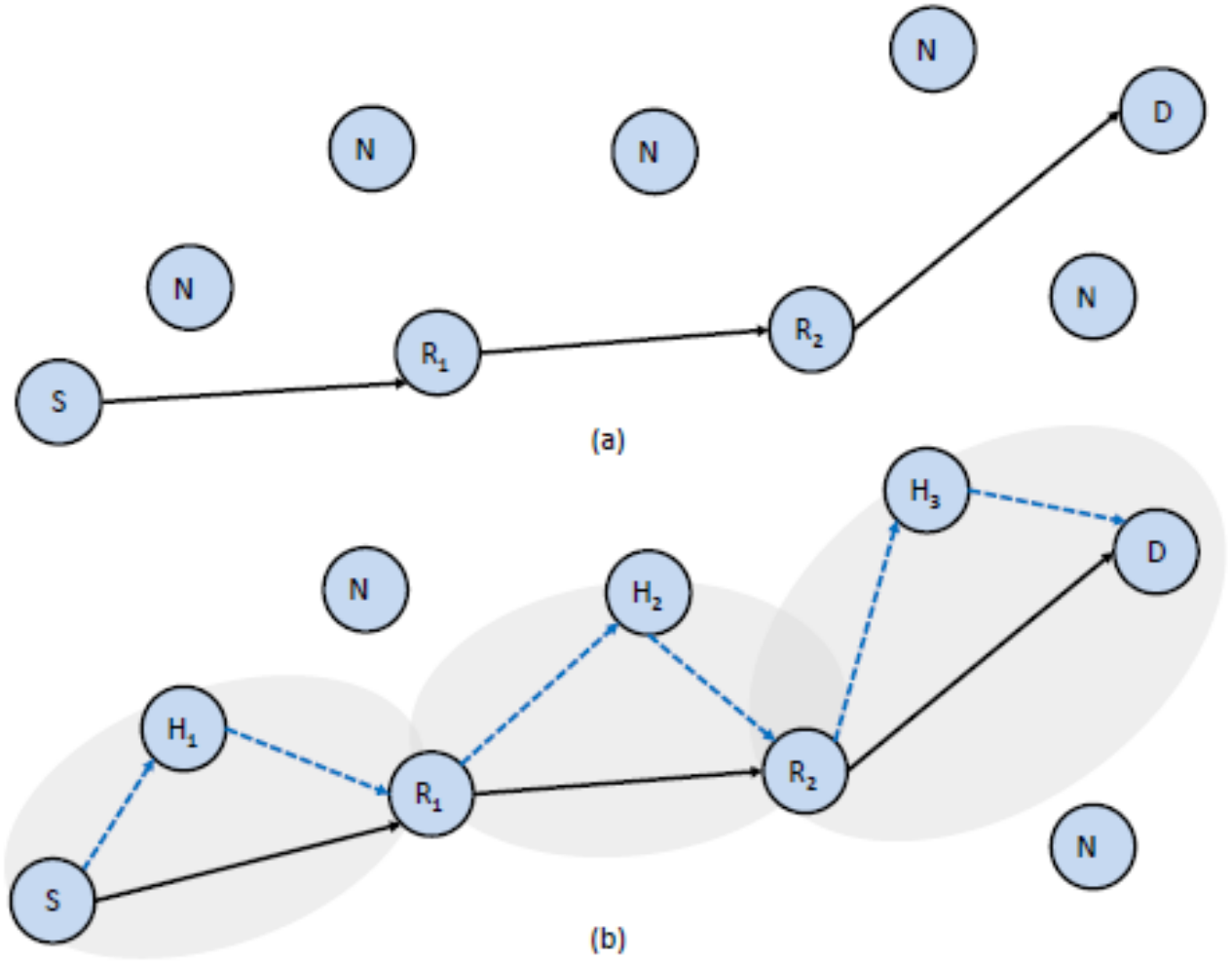


#Relays	1	2	3	4	5	6	7	8	9
Repeater									
Recoder									
Gain									

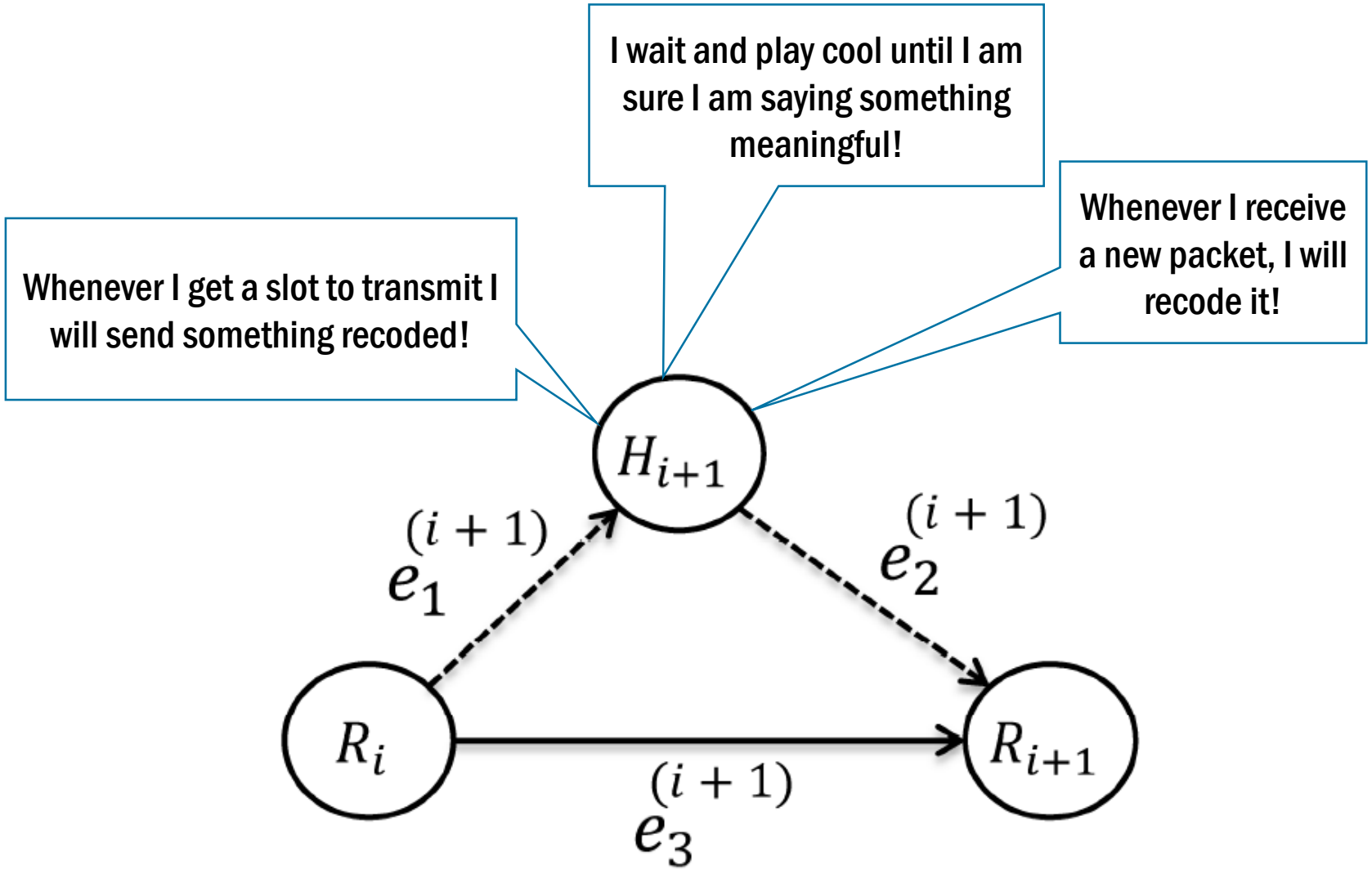
Impact of the protocol design

P. Pahlavani, D.E. Lucani, M.V. Pedersen, and F.H.P Fitzek, “PlayNCool: Opportunistic Network Coding for Local Optimization of Routing in Wireless Mesh Networks,” in Globecom 2013 Workshop - First International Workshop on Cloud-Processing in Heterogeneous Mobile Communication Networks - GLOBECOM 2013, Dec. 2013.

Wireless Mesh

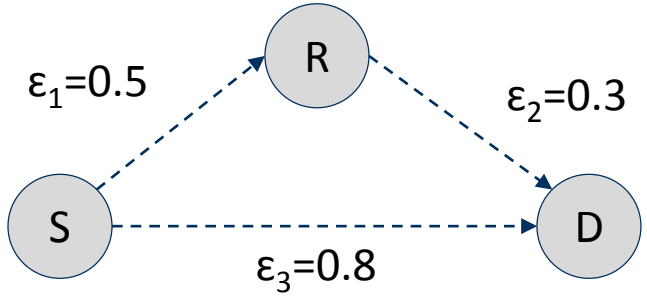


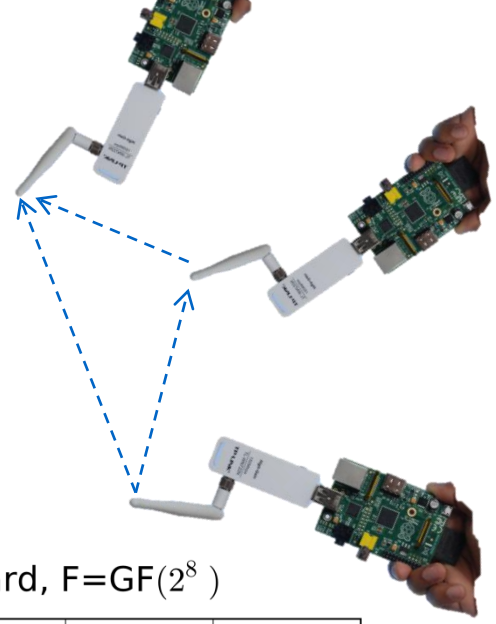
Strategies under Investigation

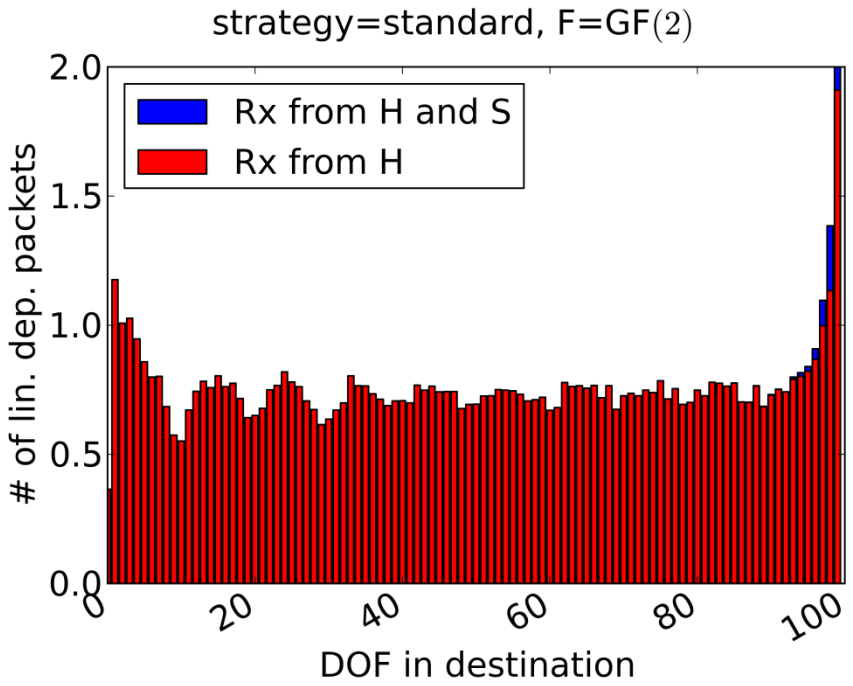


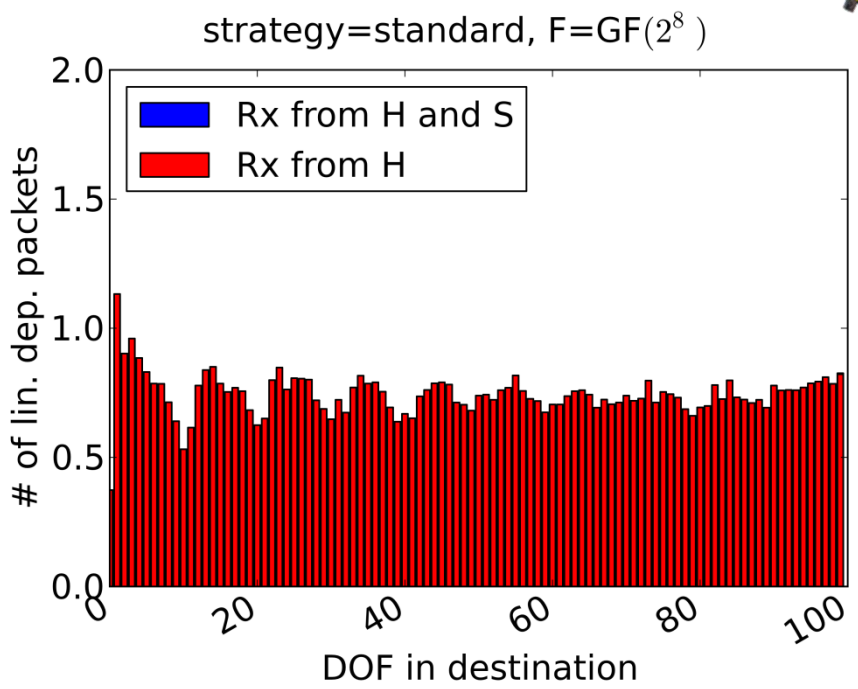
RLNC in real meshed

Agnostic recoding



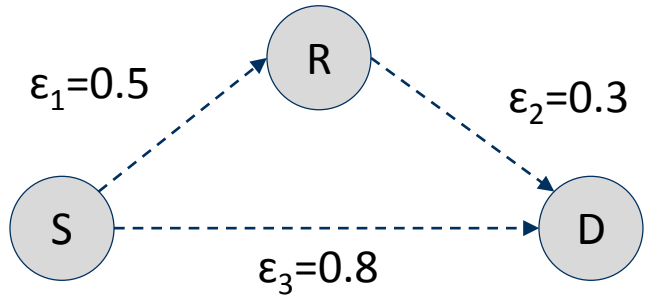


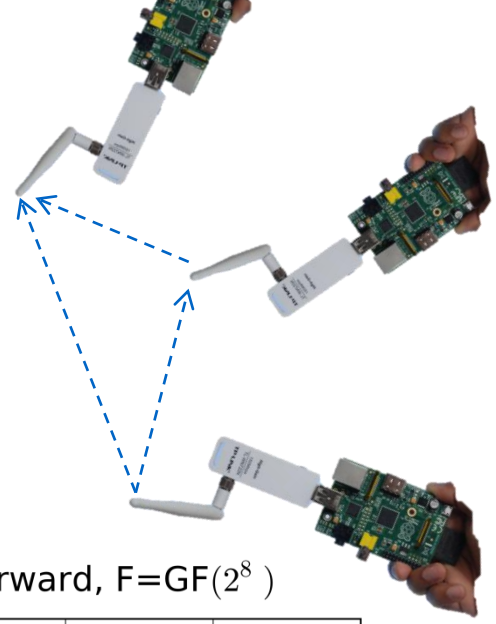


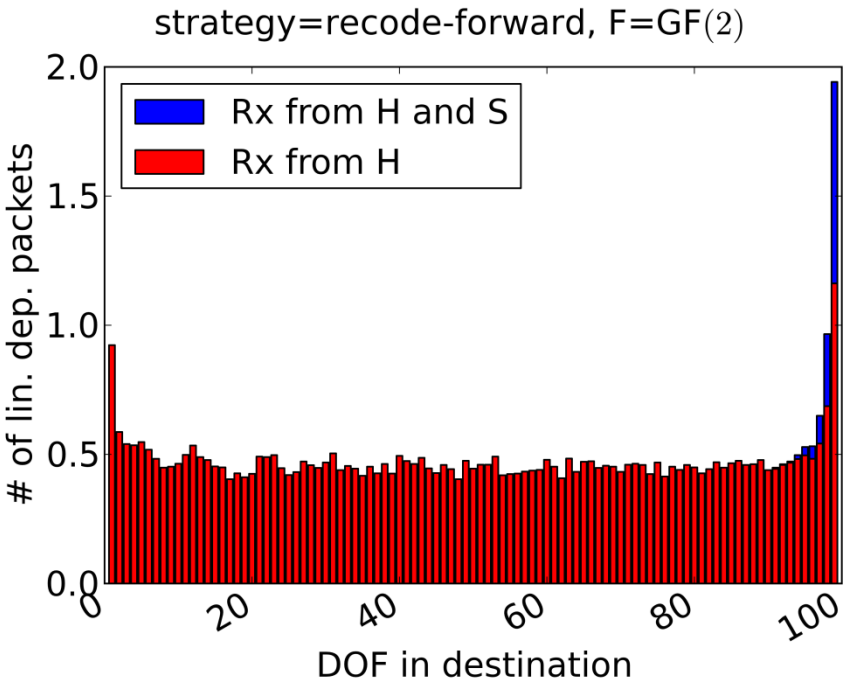


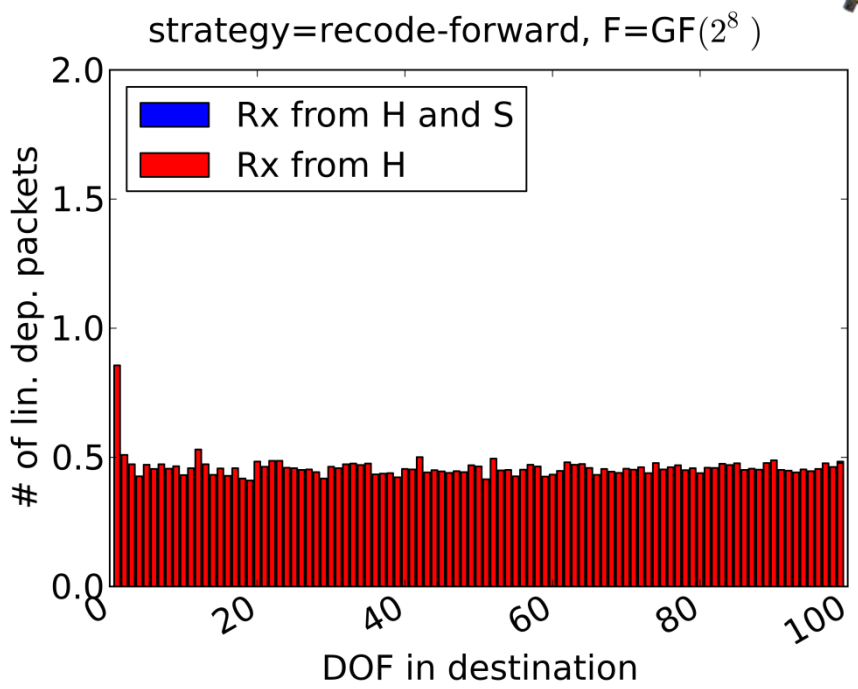
RLNC in real meshed

Rate-sensitive recoding



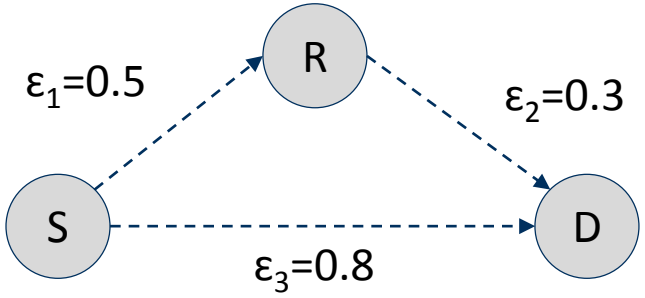


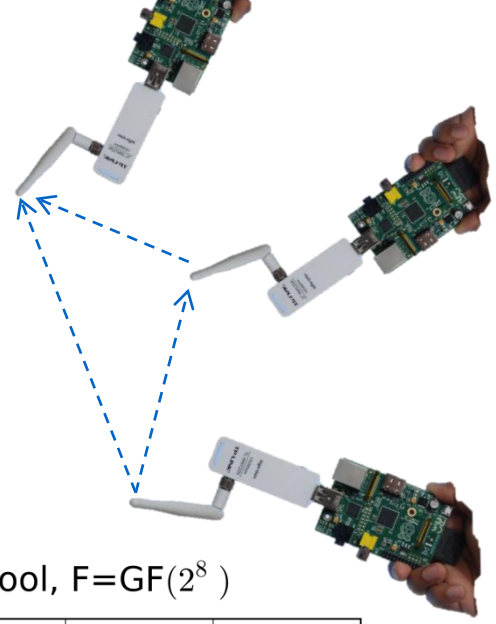


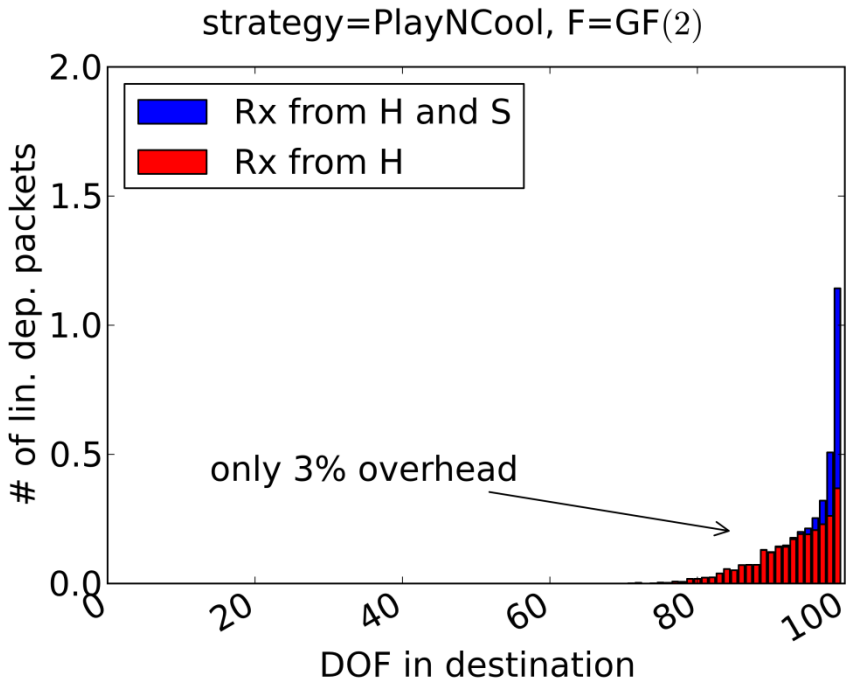


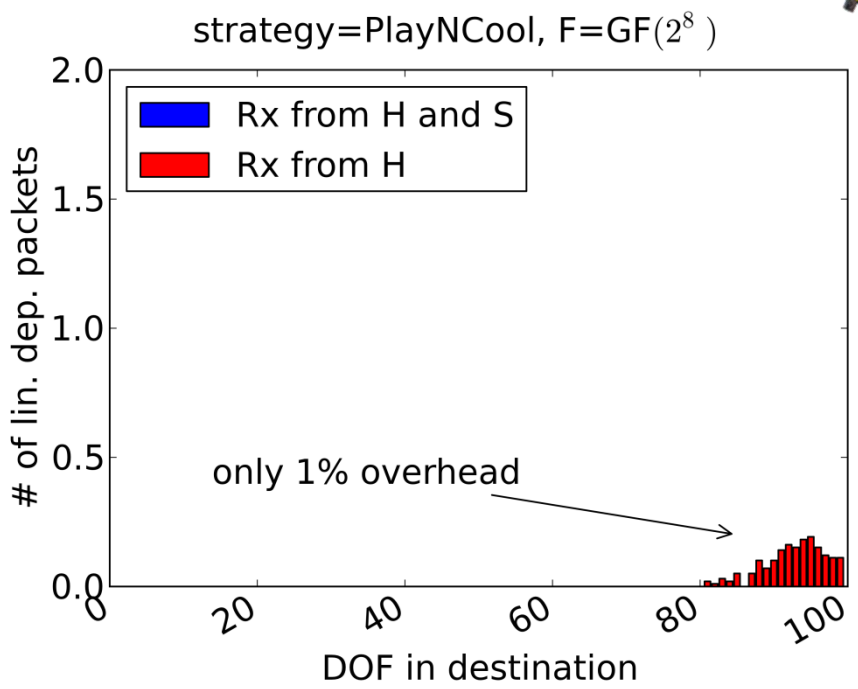
RLNC in real meshed

PlayN Cool

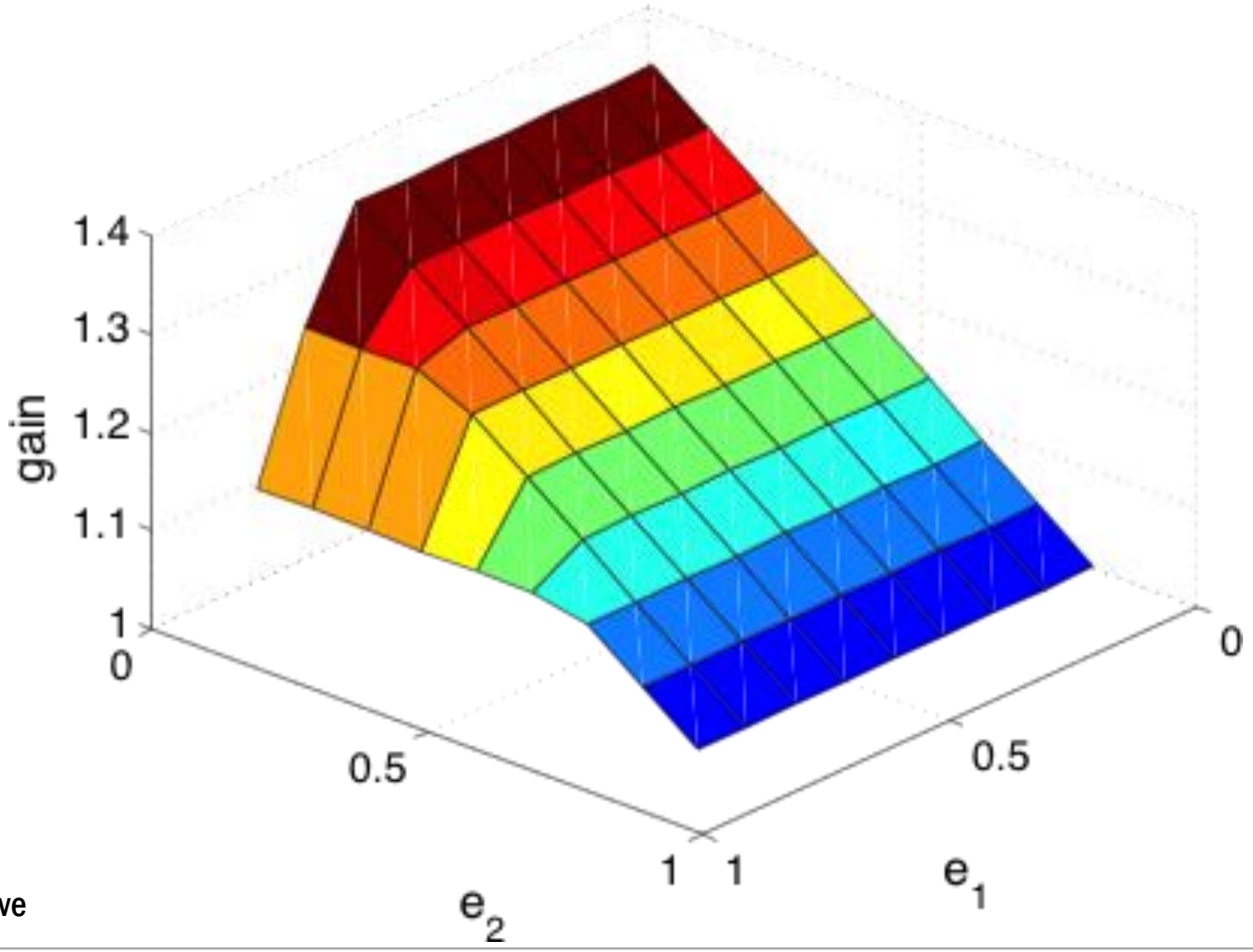






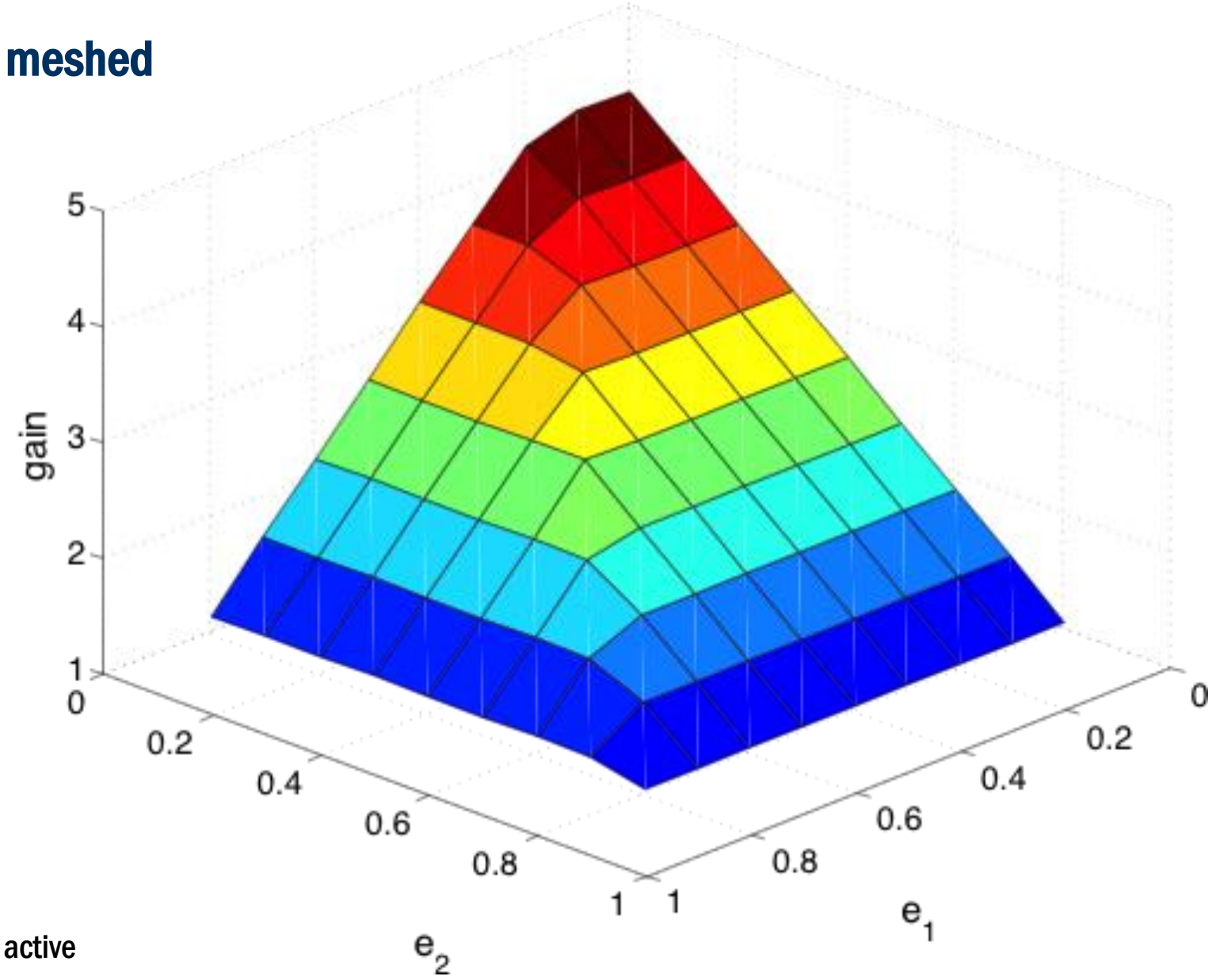


RLNC in real meshed



e3=0.30, 4 neighbors active

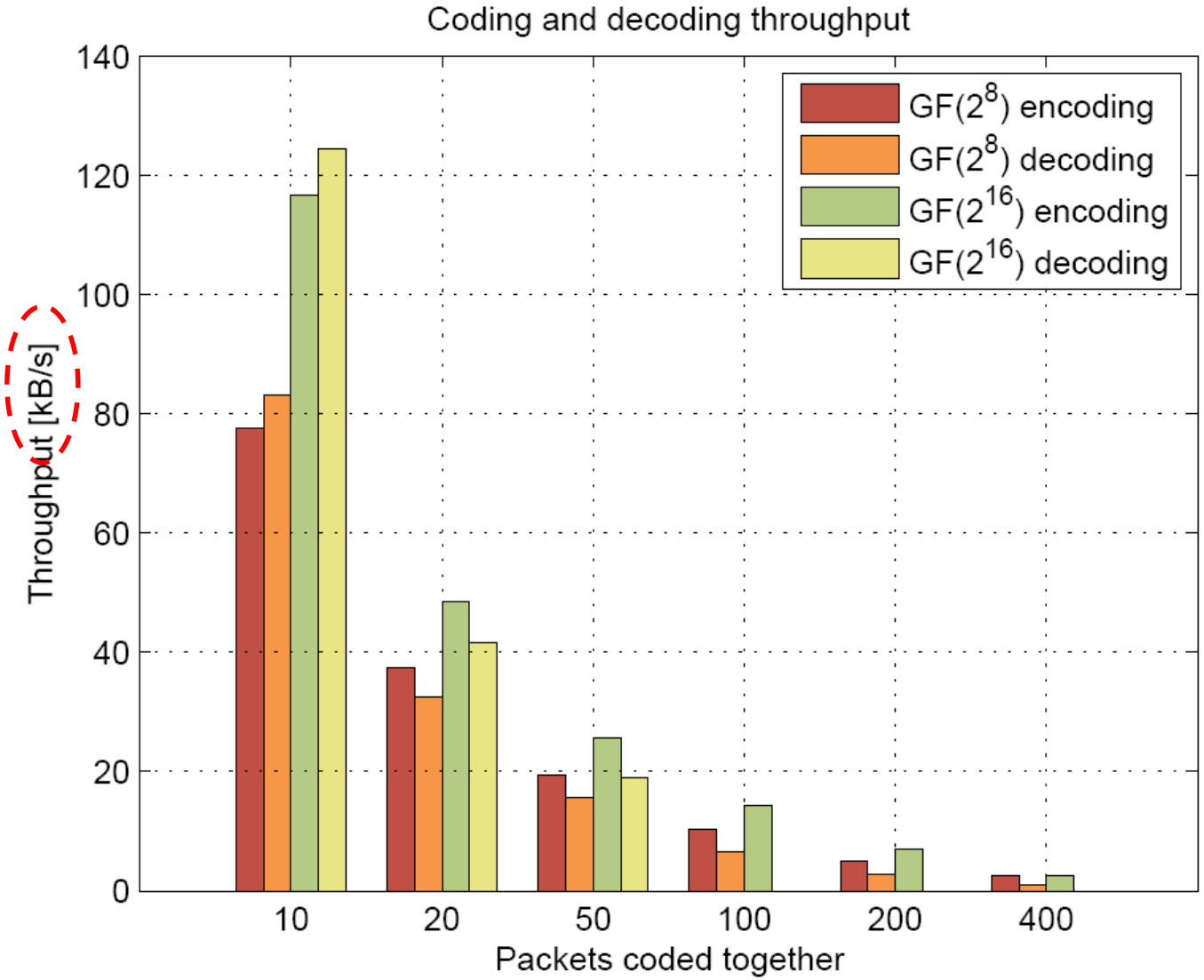
RLNC in real meshed



e3=0.80, 4 neighbors active

A Practical Guide to RLNC Libraries

S60 Implementation RLNC (2007)



Pre-allocated memory, generated the encoding vectors, so that we only had the raw encoding

Key Technologies to Speed Up

- New software design
- Right choice of G and F
  - Binary case results in low complexity
- Hardware implementation
  - Dedicated hardware (OPENGL, SIMD)
  - Multi core / Many core (HAEC)
  - Kernel
- Sparse coding & Systematic coding
- Optimal Prime Fields (OPF), e.g., 232-5