Deutsche Telekom Chair of Communication Networks
Technische Universität Dresden

Practical Implementations of Network Coding

Frank Fitzek // Summer Semester 2019

Random Linear Network Coding

Coding History

Evolution of Coding

Free
Proprietary close to patent expiry
Proprietary with long patent life

1950s
1960s
1990s
1998
2003
2014

Block Codes

Convolutional Codes

Modern Codes

- LDPCs – patent expired
- Turbo Codes – patents expired or expiring

Rateless Codes

Raptor and related codes

- Rate-less (refinement to free E2E)
- Still E2E, still static

codeon

Fulcrum Codes

codeon

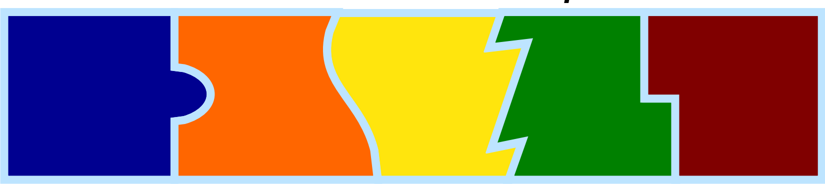
Steinwurf

- RLNC-enabled
  - *Fluid complexity* (flexible field size)
  - Breaks performance-overhead trade-off

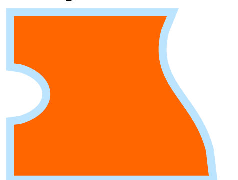
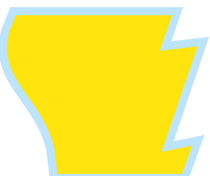
Coding Motivation

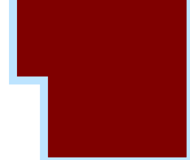
The Technology: Traditional Approach

data broken into five pieces

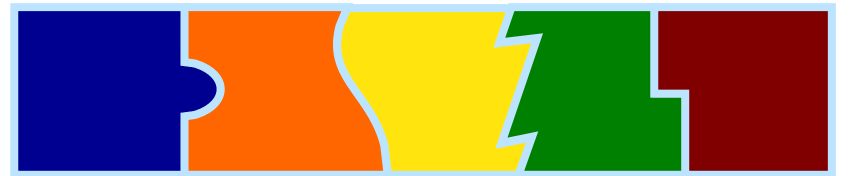


data conveyed / stored on five different paths / clouds

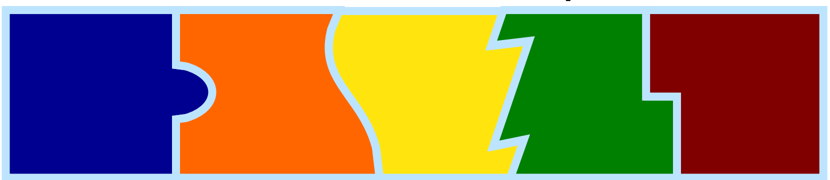


data retrieved from five different paths / clouds → success

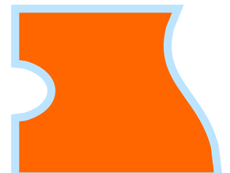


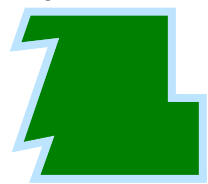
The Technology: Traditional Approach - Problem

data broken into five pieces



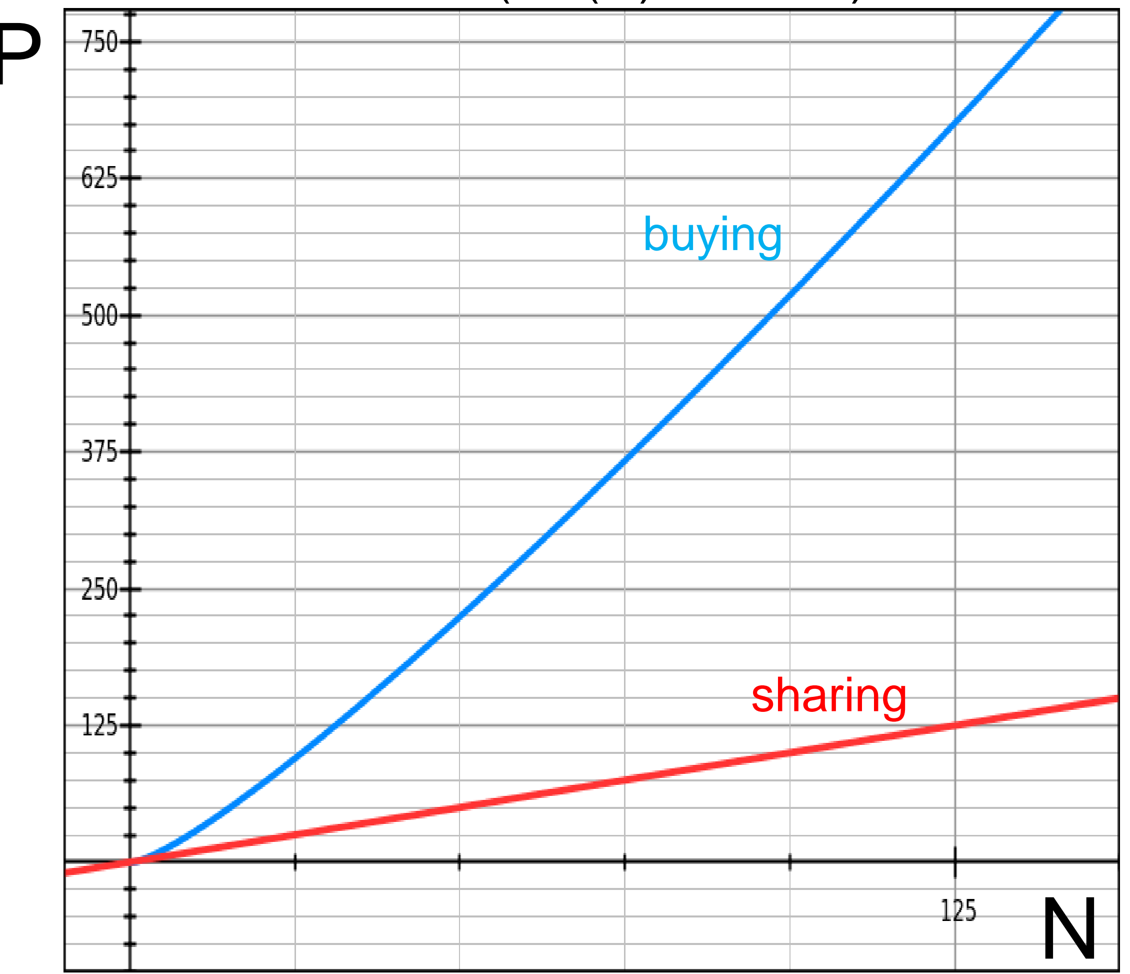
data conveyed / stored on five different paths / clouds





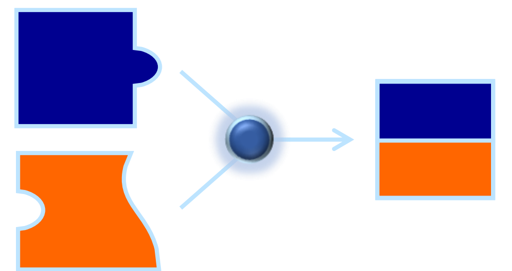
data retrieved from five ~~different~~ paths / clouds → failure

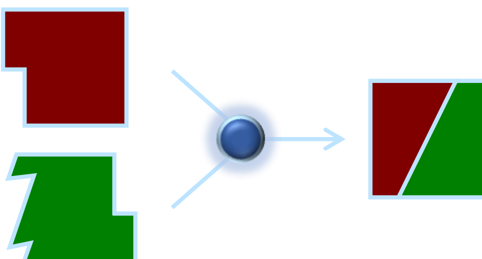
Coupon Collector's Problem

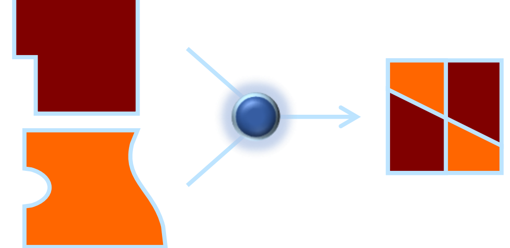


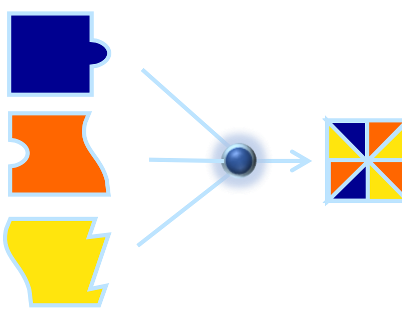
The Technology: Coding

coding data





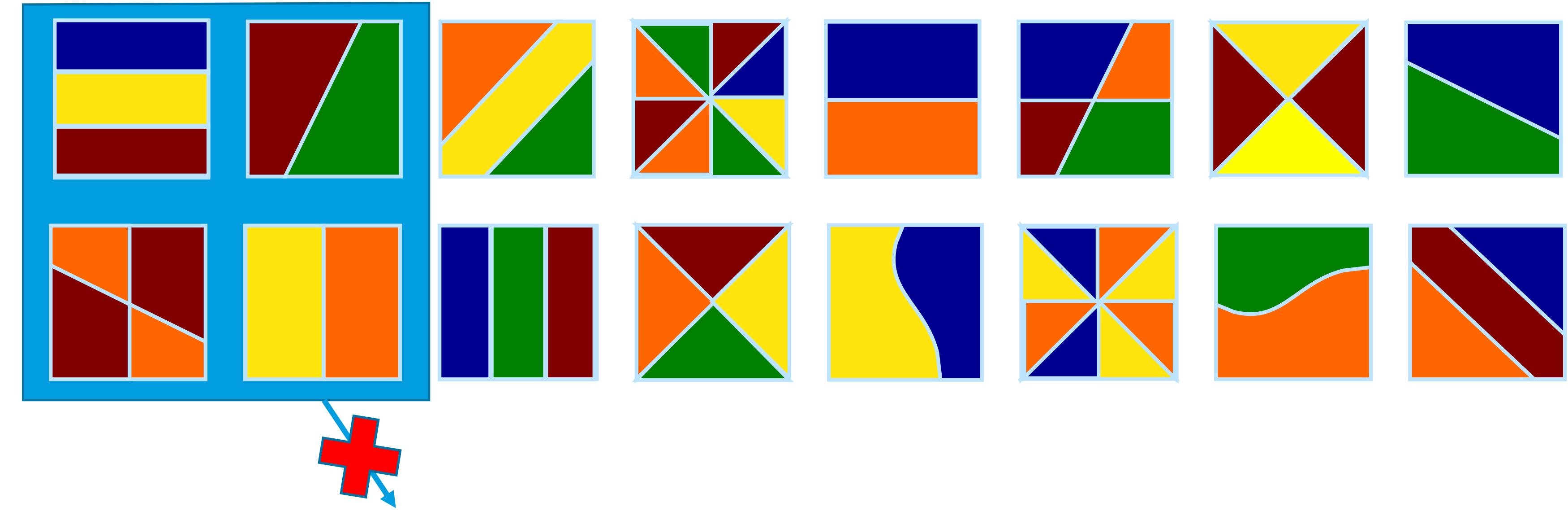




The Technology: Decoding

The Technology: Decoding

The Technology: Decoding



The Technology: Decoding

The Technology: Decoding

Will any five combinations end up in a successful decoding?

Coding for Multipath – Multicloud

Any mixtures in any order will do

Coding as an Additional Security Measure

Coded Packets

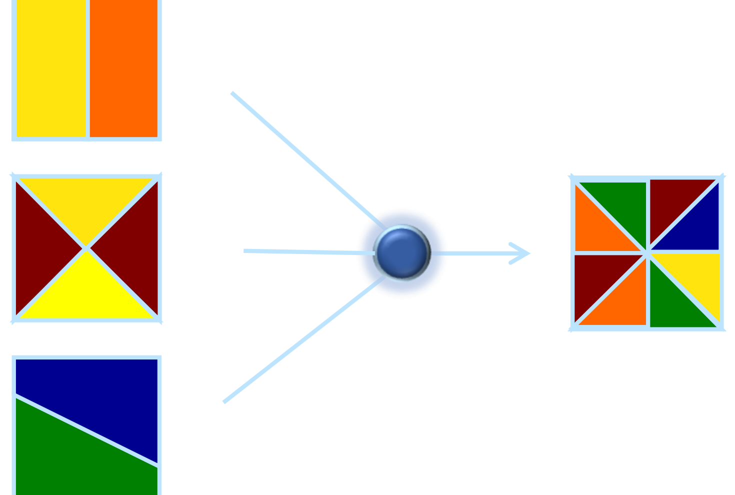
Native Packets

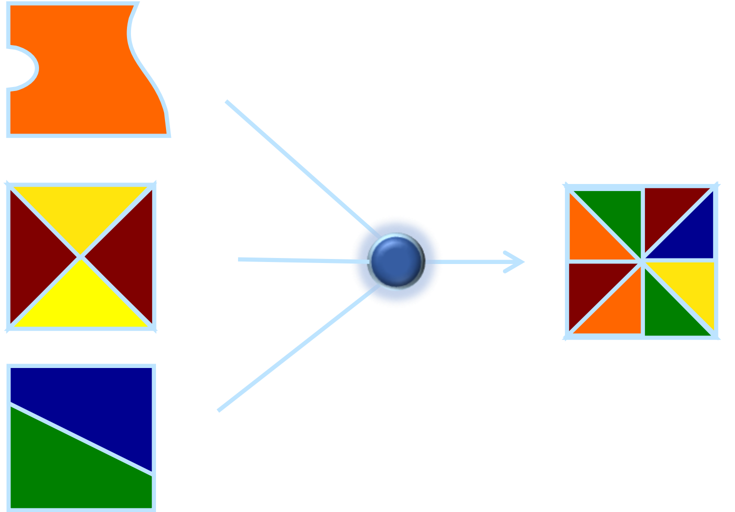
Multipath Transport

Distributed Storage

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

The Technology: Recoding





The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

The Technology: Recoding

Why does recoding matter?

Why does recoding matter? – Traditional Coding → Success

Why does recoding matter? – Traditional Coding → Failure

Why does recoding matter? Traditional Coding → Failure

Why does recoding matter? Traditional Coding → Failure

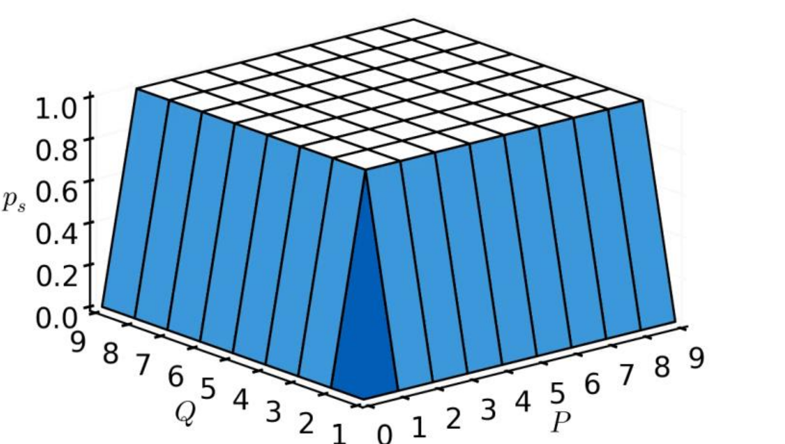
We code more packets together!

Duplicates!

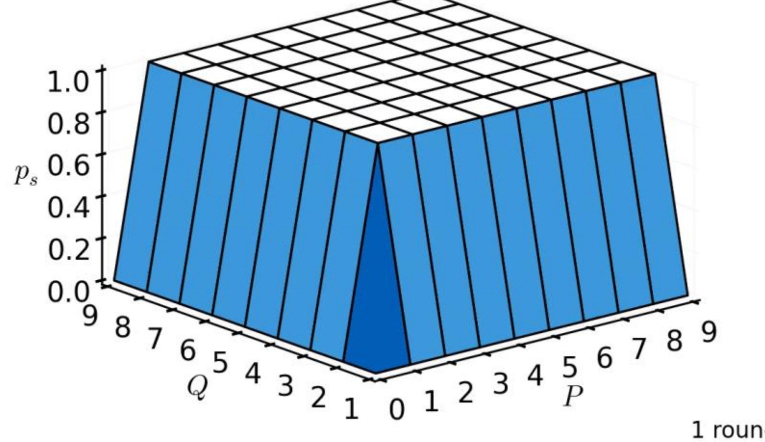
Why does recoding matter? Recoding → Success

Comparison

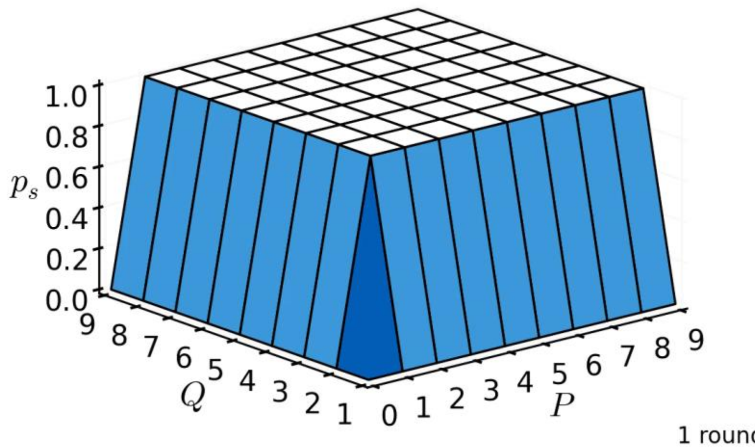
Uncoded



RS coded



Recoding



see lecture on storage

Random Linear Coding

Random Linear Network Coding

Orignal packets

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

Random Linear Network Coding

coding
coefficients

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

Random Linear Network Coding

coded packets

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

Random Linear Network Coding

coded packets

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

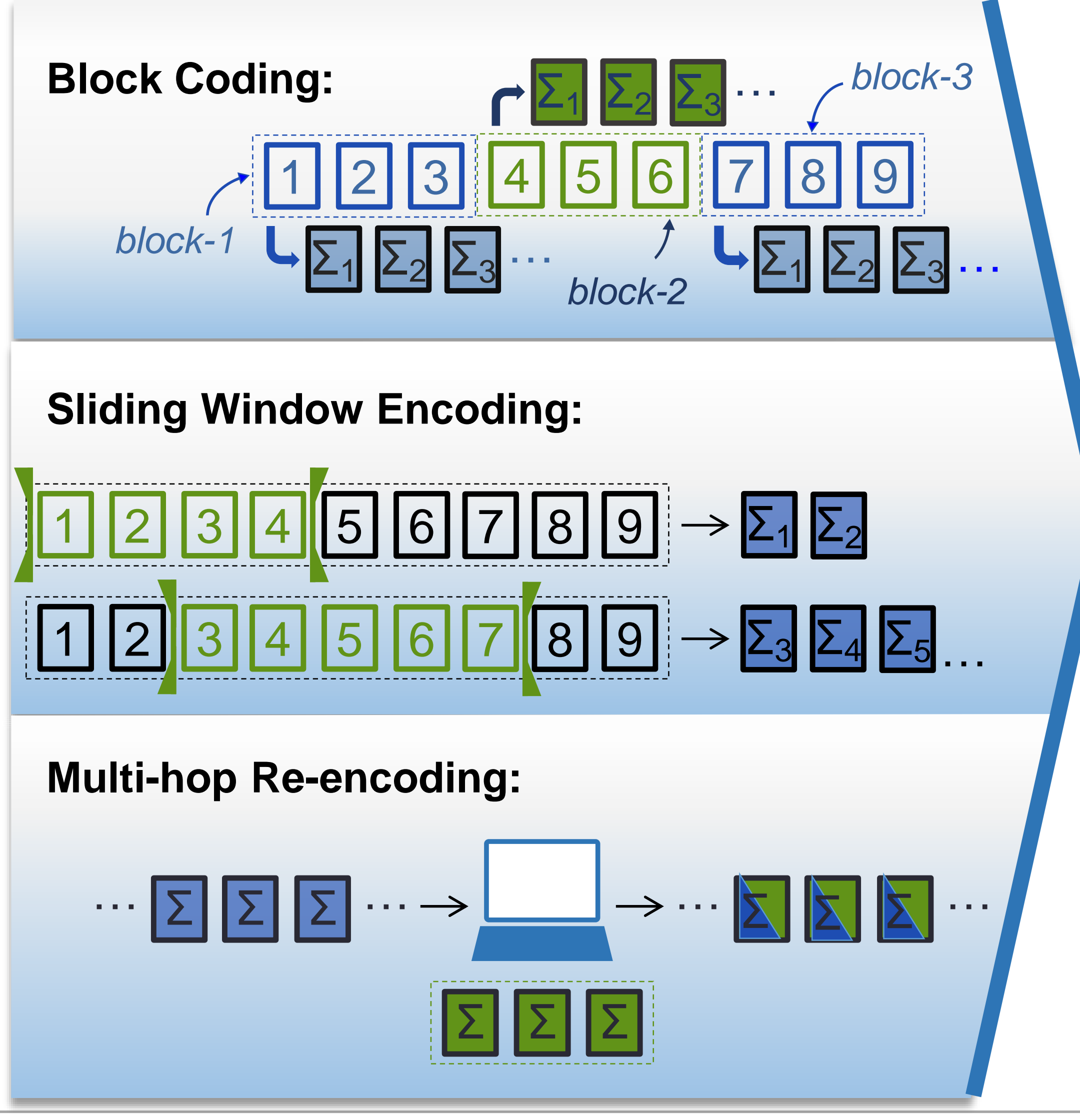
Gaussian elimination *n* x *n matrix* requires
$An^3 + Bn^2 + Cn$ operations.

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)

RLNC: The Technology



decoding scheme

(simple equation-solving)

$$1 = a'_1 \Sigma_1 + b'_1 \Sigma_2 + g'_1 \Sigma_3$$

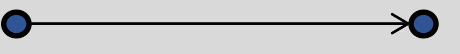
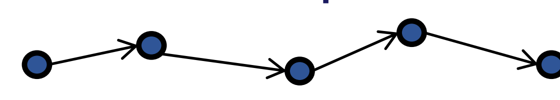
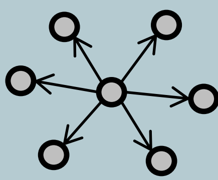
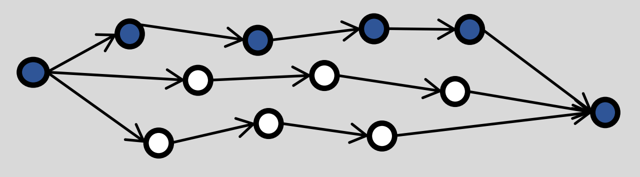
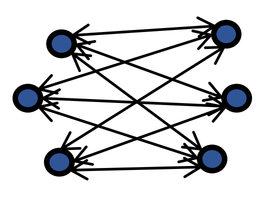
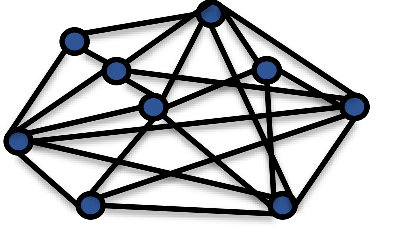
$$2 = a'_2 \Sigma_1 + b'_2 \Sigma_2 + g'_2 \Sigma_3$$

$$3 = 1 + b'_3 \Sigma_2 + g'_3 \Sigma_3$$

obtained through Gaussian Elimination

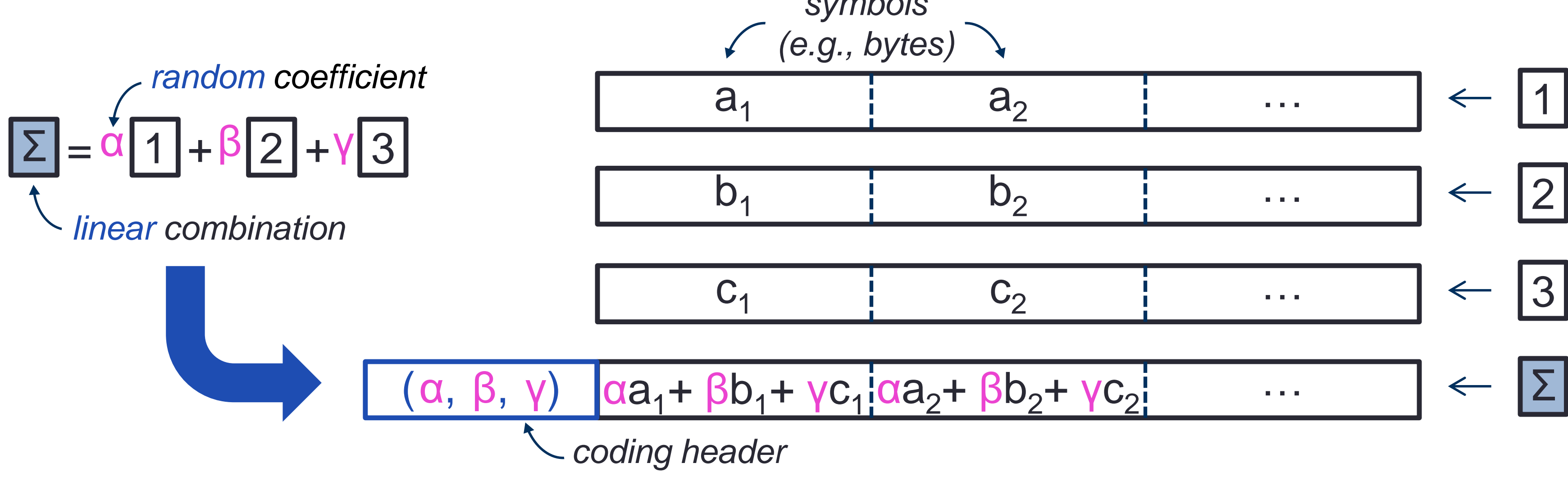
Can decode using both encoded and un-encoded packets

RLNC: The Technology

Coding Today (all End-to-End)	Coding Tomorrow with RLNC	
	Classical + Sliding Window Encoding 	Real time video streaming, TCP, SDN…
Classical	**Multihop** 	Edge caches, wireless mesh, reliable multicast, satellites, small relay topologies, SDN…
Multicast 	**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) …

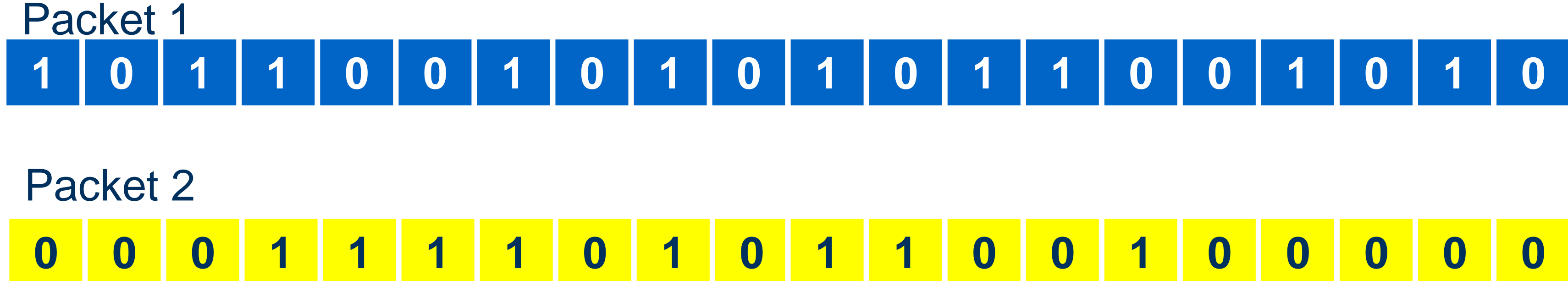
Coding packets 101

How does RLNC work?

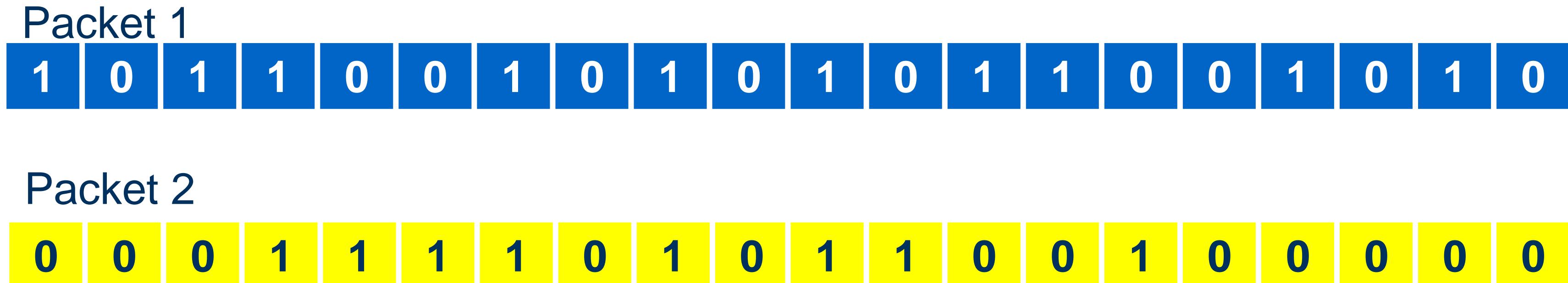


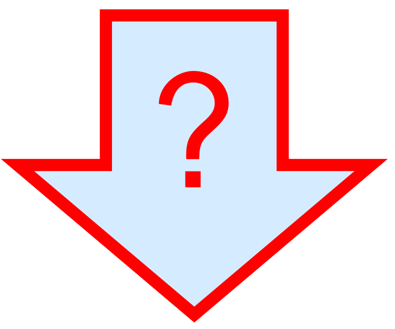
- Random generation of coefficients
- Code embedded within data
- No state tracking
- ***Versatile*** code

Coding packets 101



Coding packets 101





Coded Packet (initially empty)

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

XOR

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

XOR

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

XOR

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

XOR

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

XOR

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

Packet 2

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

symbol

XOR

What about the encoding vector in this example?

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

$\alpha=1$

Packet 2

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

symbol

$\beta=1$

XOR

What about the encoding vector in this example?

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

$\alpha=1$

Packet 2

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

symbol

$\beta=1$

XOR

What about the encoding vector in this example?

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

Coded Packet

Coding packets 101: Field Size GF(2)

Packet 1

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

$\alpha=1$

Packet 2

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

symbol

$\beta=0$

XOR

What about the encoding vector in this example?

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

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

$\alpha=10$

Packet 2

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

symbol

$\beta=01$

GF(2^2)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Coded Packet

Coding packets 101: Field Size $GF(2^2)$

Packet 1

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

α=10

Packet 2

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

10*10 XOR 00*01

$GF(2^2)$

β=01

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

$\alpha=10$

Packet 2

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

$\beta=01$

GF(2^2)

10*10 XOR 00*01

	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

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Coded Packet

Coding packets 101: Field Size $GF(2^2)$

Packet 1

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

$\alpha=10$

Packet 2

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

$\beta=01$

$GF(2^2)$

10*10 XOR 00*01

11 XOR

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

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

α=10

Packet 2

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

β=01

GF(2^2)

10*10 XOR 00*01

11 XOR 00

11

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

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

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

α=10

Packet 2

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

β=01

GF(2^2)

10*10 XOR 00*01

11 XOR 00

11

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

10*10=100
100 mod 111=1 R 11

Carry-less multiply
+ modulo
Irreducible polynom 111

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

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

α=10

Packet 2

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

GF(2^2)

β=01

10*10 XOR 00*01

11 XOR 00

11

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

11*11
11
11_
101

11*11=101
101 mod 111=1 R 10
111
10

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

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

$\alpha=10$

Packet 2

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

$\beta=01$

GF(2^2)

10*10 XOR 00*01

11 XOR 00

11

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

11*11
11
11_
101

11*11=101
101 mod 111=1 R 10
111
10

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

Coded Packet

Coding packets 101: Field Size GF(2^2)

Packet 1

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

$\alpha=10$

Packet 2

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

$\beta=01$

GF(2^2)

Try it out!
Result is correct!

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

Coded Packet

Coding packets 101: Field Size GF(2^4)

Packet 1

1	0	1	1	0	0	1	0	1

α=1100

Packet 2

0	0	0	1	1	1	1	0	1

GF(2^4)

β=0001

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

Coded Packet

In [8]:

```
import fifi

field = fifi.simple_online_binary4()
order = 2**4

table = ''


for i in range(order):
    for j in range(order):
        table += '{:02d} '.format(field.multiply(i,j))
    table += '\n'

print(table)
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 02 04 06 08 10 12 14 03 01 07 05 11 09 15 13
00 03 06 05 12 15 10 09 11 08 13 14 07 04 01 02
00 04 08 12 03 07 11 15 06 02 14 10 05 01 13 09
00 05 10 15 07 02 13 08 14 11 04 01 09 12 03 06
00 06 12 10 11 13 07 01 05 03 09 15 14 08 02 04
00 07 14 09 15 08 01 06 13 10 03 04 02 05 12 11
00 08 03 11 06 14 05 13 12 04 15 07 10 02 09 01
00 09 01 08 02 11 03 10 04 13 05 12 06 15 07 14
00 10 07 13 14 04 09 03 15 05 08 02 01 11 06 12
00 11 05 14 10 01 15 04 07 12 02 09 13 06 08 03
00 12 11 07 05 09 14 02 10 06 01 13 15 03 04 08
00 13 09 04 01 12 08 05 02 15 11 06 03 14 10 07
00 14 15 01 13 03 02 12 09 07 06 08 04 10 11 05
00 15 13 02 09 06 04 11 01 14 12 03 08 07 05 10
```

Coding packets 101: Field Size GF(2^8)

Packet 1

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

α=10101010

Packet 2

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

β=01010101

GF(2^8)

Lookup table 2^8*2^8*8bit= 65536 bytes (Still ok)

Lookup table 2^{16}*2^{16}*16bit= 8.6 Gbytes (online computation)

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

16 bit

Coded Packet

Coding packets 101: Field Size GF(2^8)

Packet 1

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

α=10101010

Packet 2

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

01101000

GF(2^8)

β=01010101

01001000

10101010*10110010=0100101110110100
0100101110110100 mod 100011101 = 01101000

01010101*00011110=0000011000000110
0000011000000110 mod 100011101 = 01001000

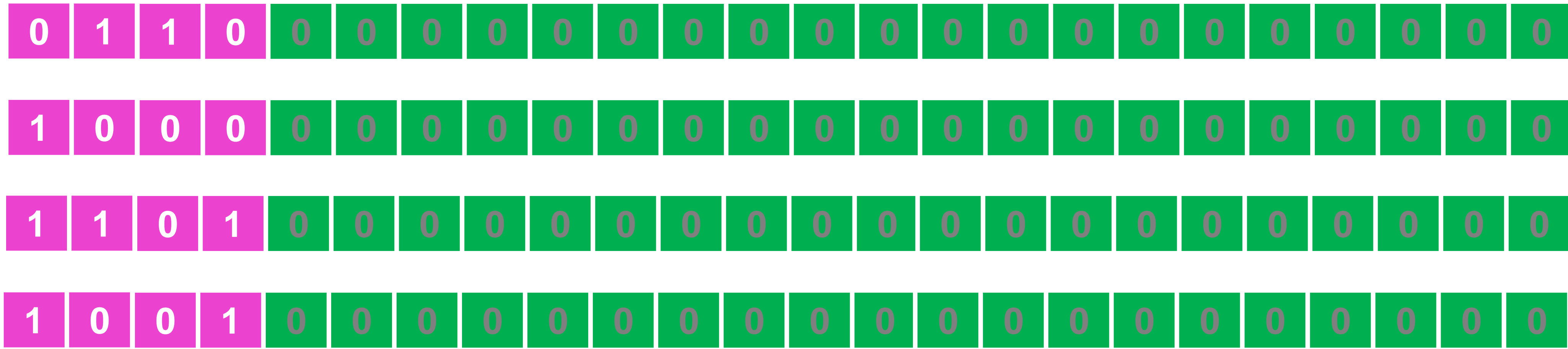
01101000 XOR 01001000 = 00100000

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

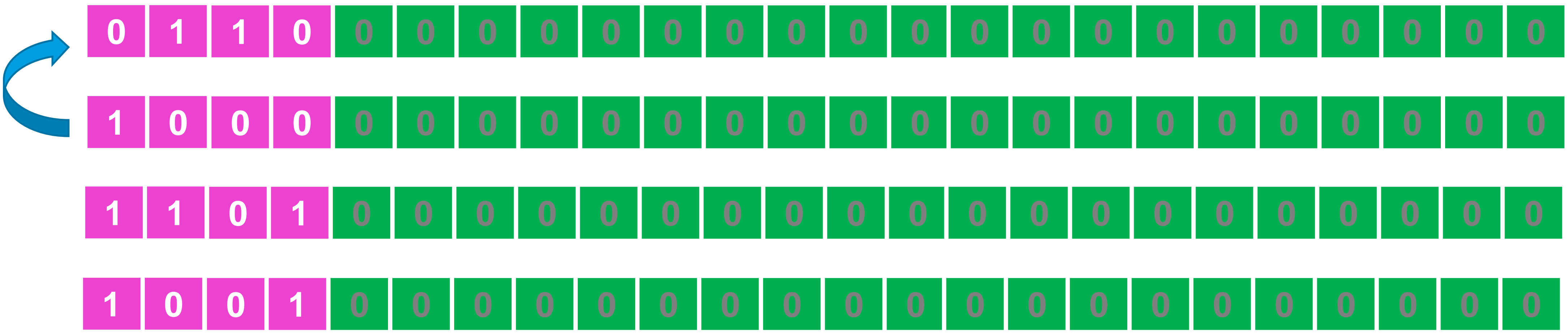
16 bit

Coded Packet

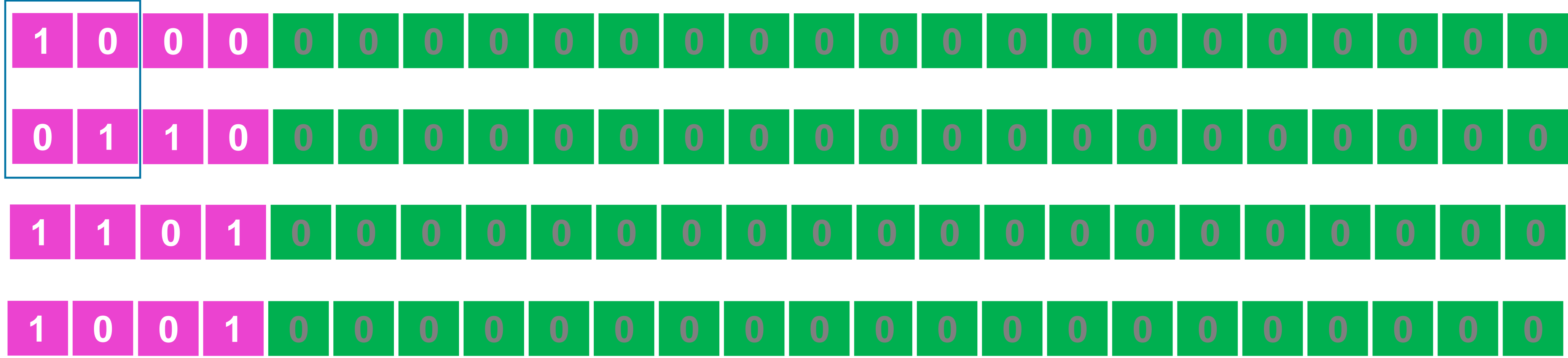
Decoding packets 101: Field Size GF(2)



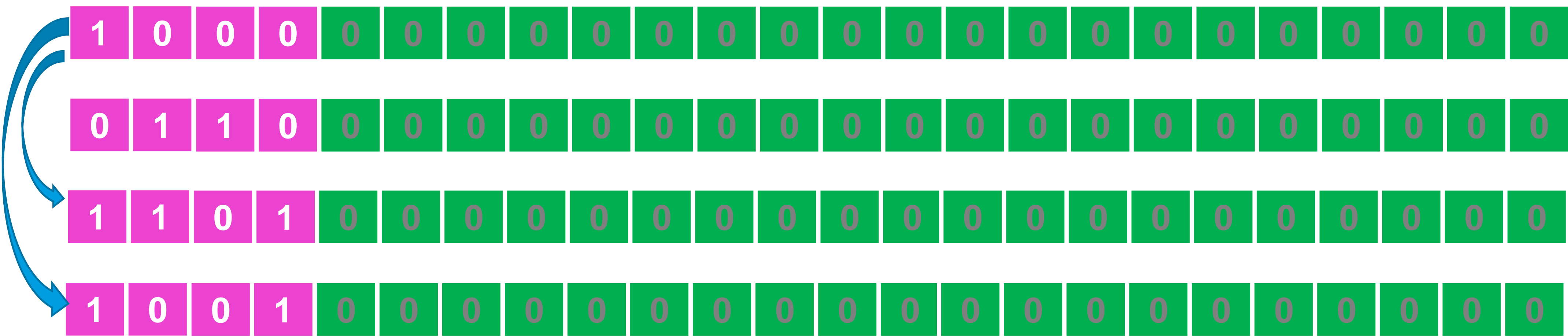
Decoding packets 101: Field Size GF(2)



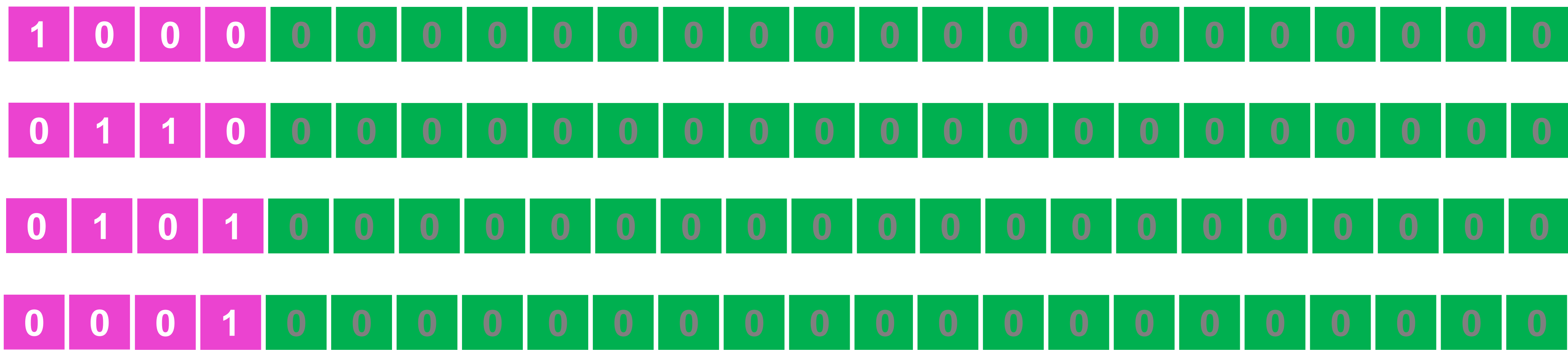
Decoding packets 101: Field Size GF(2)



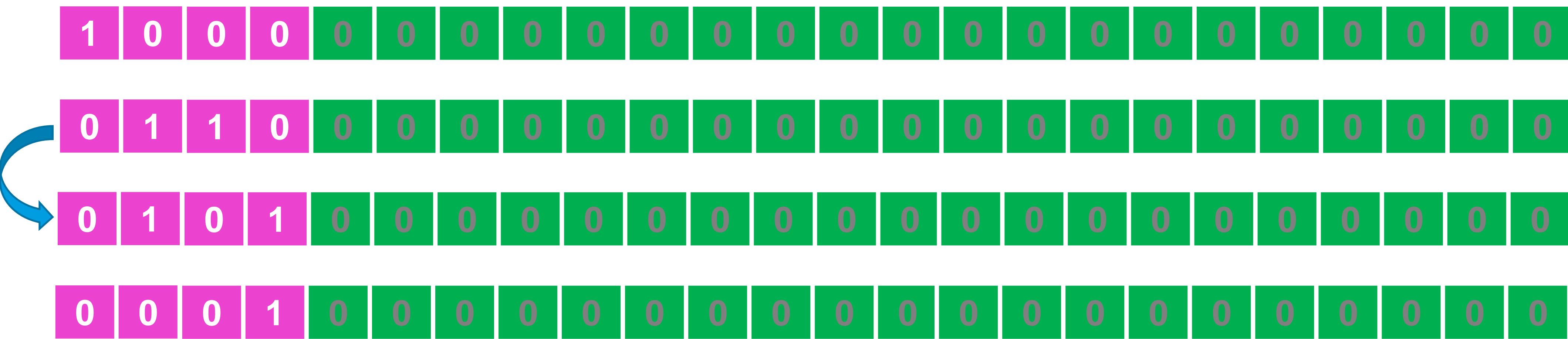
Decoding packets 101: Field Size GF(2)



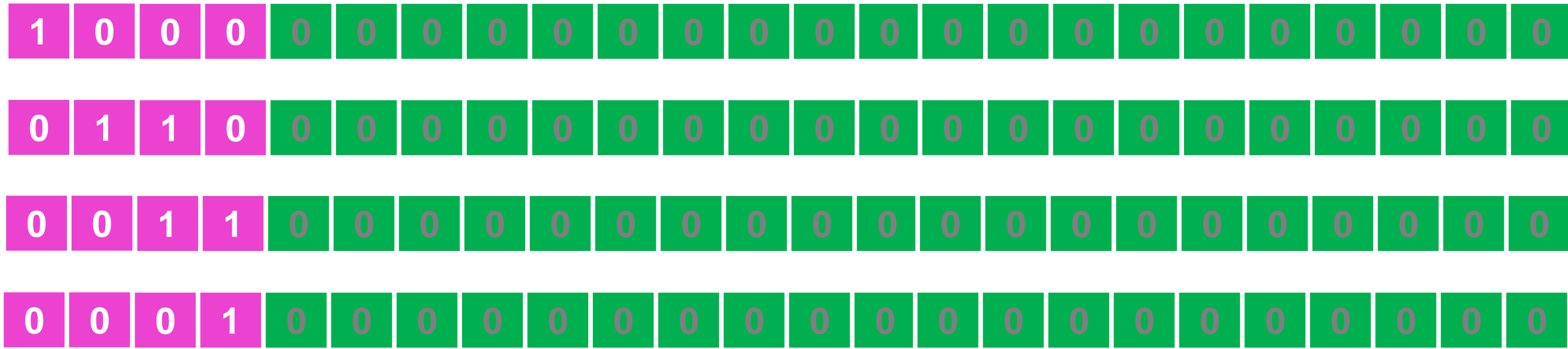
Decoding packets 101: Field Size GF(2)



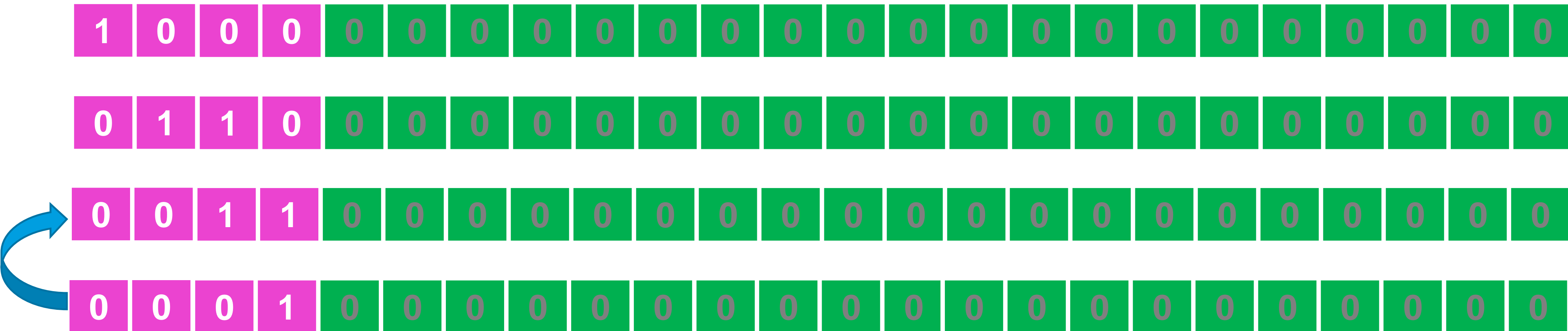
Decoding packets 101: Field Size GF(2)



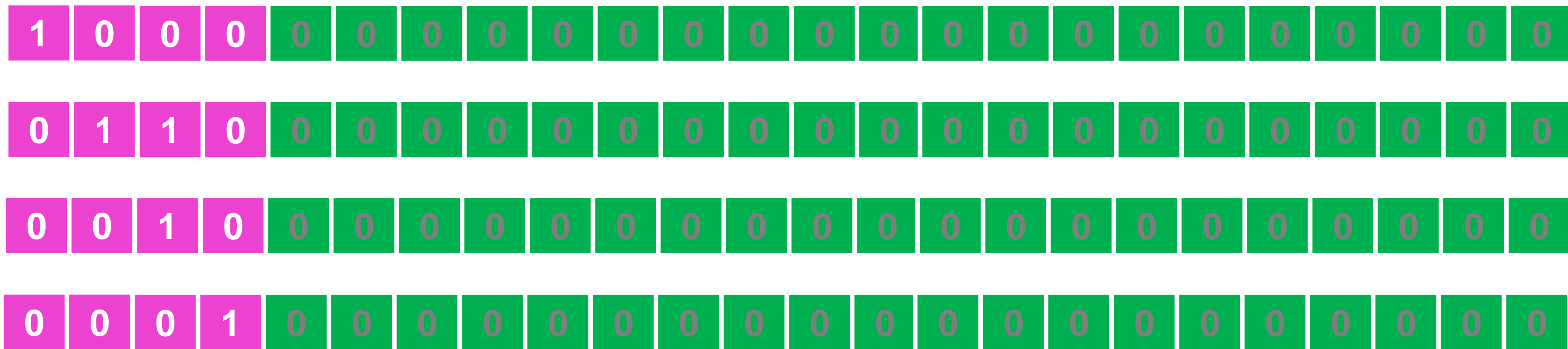
Decoding packets 101: Field Size GF(2)



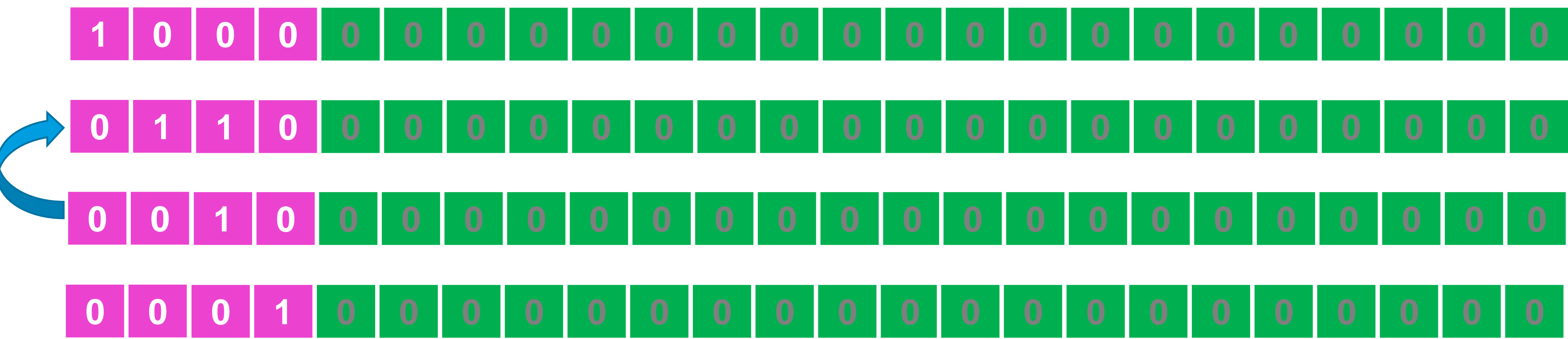
Decoding packets 101: Field Size GF(2)



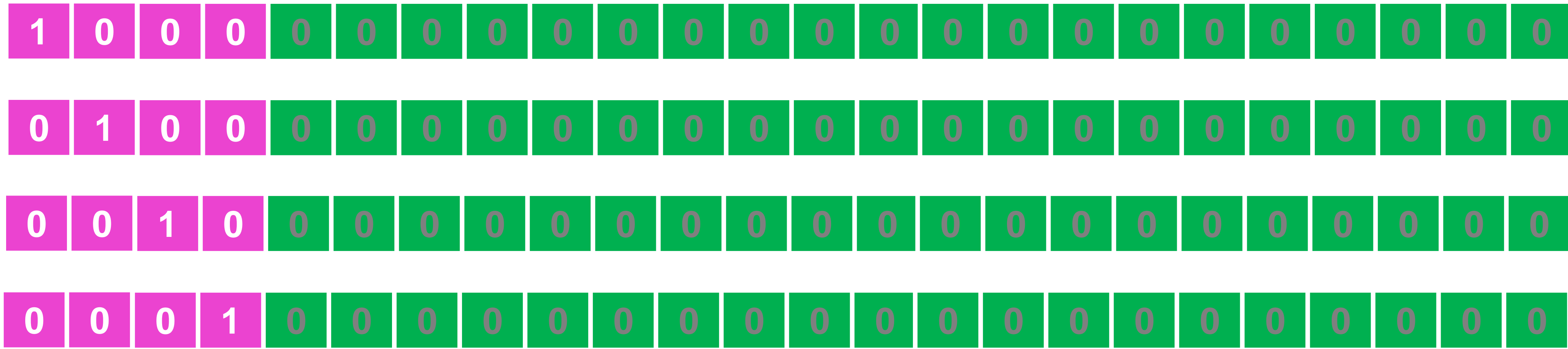
Decoding packets 101: Field Size GF(2)



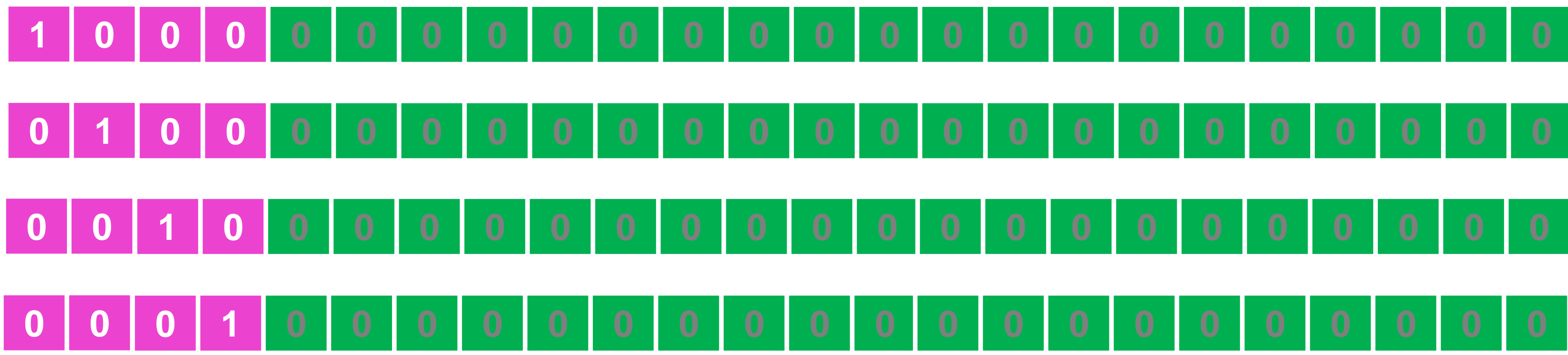
Decoding packets 101: Field Size GF(2)



Decoding packets 101: Field Size GF(2)

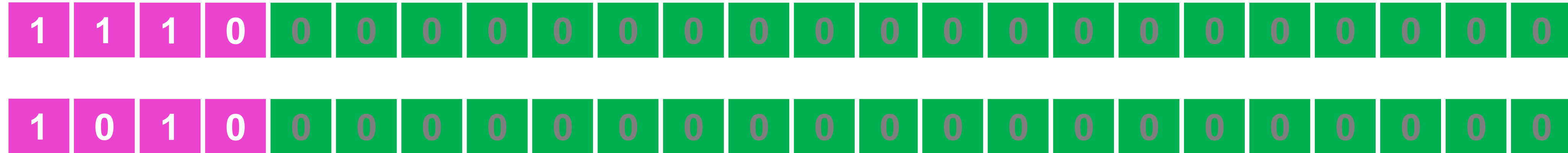


Decoding packets 101: Field Size GF(2)



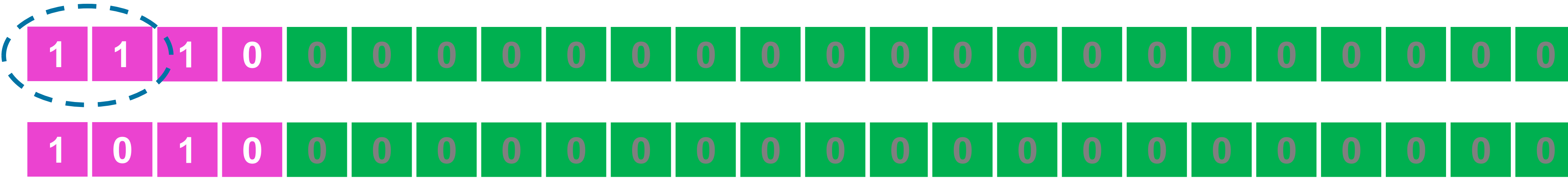
All operations on the encoding vector have to be done also on the payload.

Decoding packets 101: Field Size GF(2^2)



- Larger fields are a bit harder to decode
- Not just XOR

Decoding packets 101: Field Size GF(2^2)



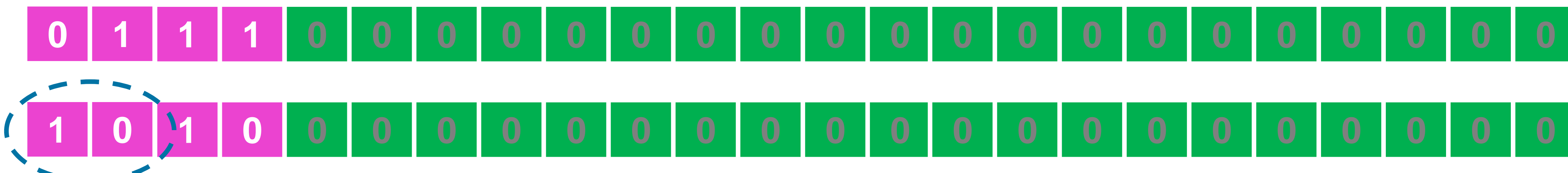
- First we look at the first symbol 11_b (3_{dec}), which needs to become 01_b (1_{dec})
- We find inverse of 11_b (3_{dec}), which is 10_b (2_{dec}), and multiply the first packet with it
- Not just the encoding vector, but also the whole packet

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Find inverse: With multiplier would give use the result 1?

~1→1 ~2→3 ~3→2

Decoding packets 101: Field Size GF(2^2)

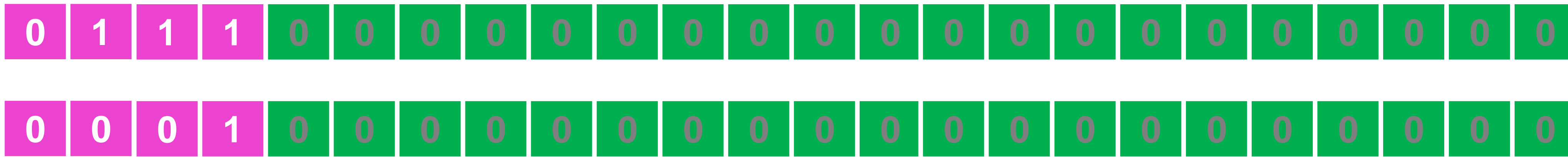


- Now we have the pivot element for the first row
- Multiply new first row with first symbol of second packet 10_b and subtract from the second row
- So multiply 01_11 with 10_b gives us 10_01
- Subtracting (XOR) 10_01 from 10_10 gives us 0011

- You could even use an inversion of packet 2 and subtract them from each other, but this would lead to an increased complexity.

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

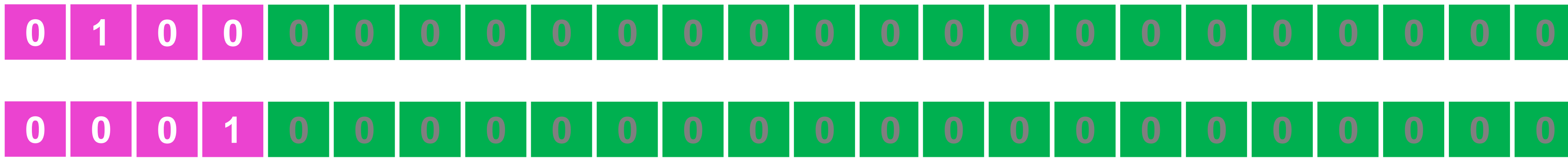
Decoding packets 101: Field Size GF(2^2)



- Packet 2 is ready (we were lucky)
- The last symbol of packet 2 was already 01_b, if not you multiply the packet with the inverse of the symbol
- Now we use packet 2 to clean up backwards
- Multiply packet 2 with second symbol of packet one and subtract from packet 1
- EV 00_01 becomes EV 00_11
- Subtract both from each other

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

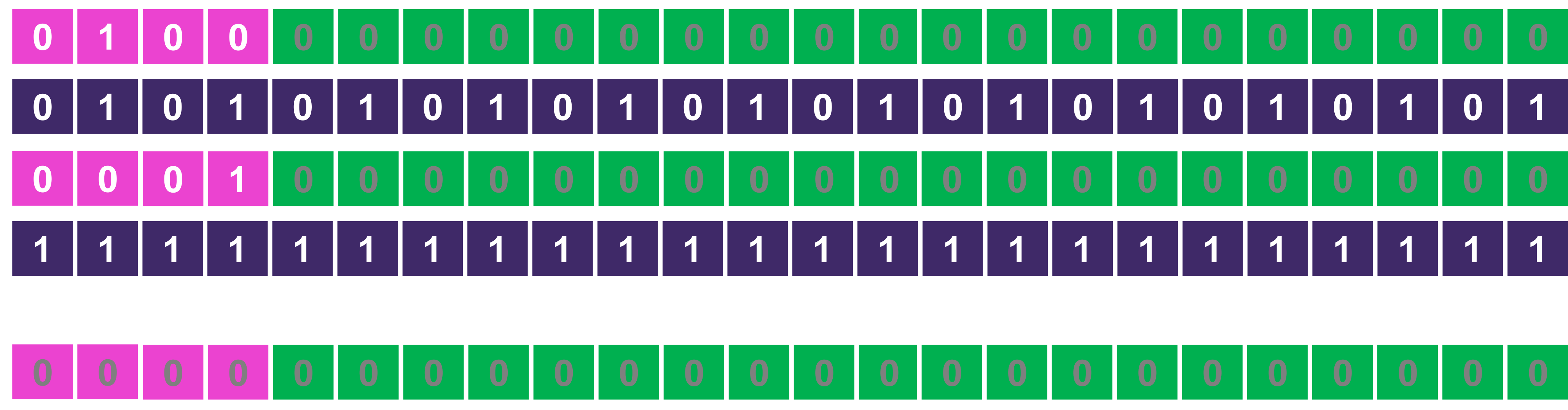
Decoding packets 101: Field Size GF(2^2)



- Done
- And every operation has to be performed on the payload as well

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

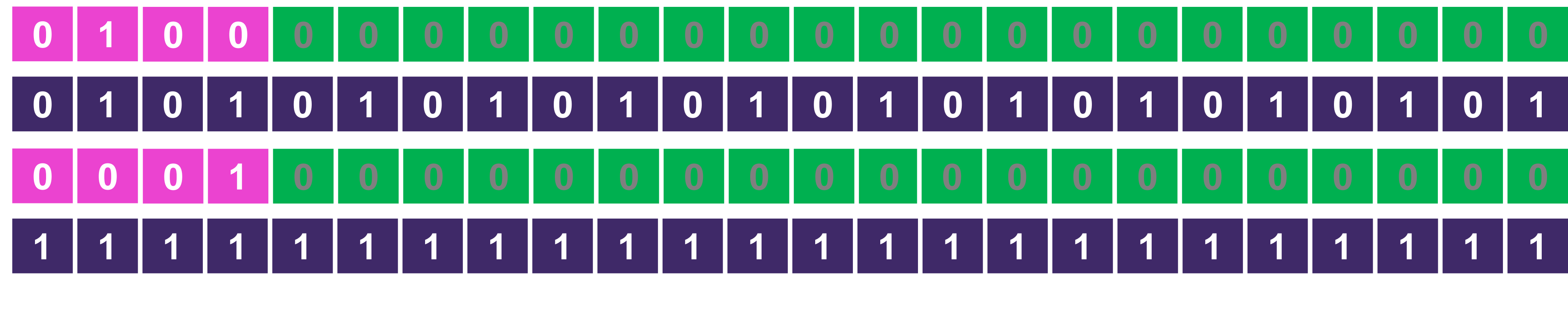
Recoding packets 101: Field Size $GF(2^2)$



- How is recoding done?

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

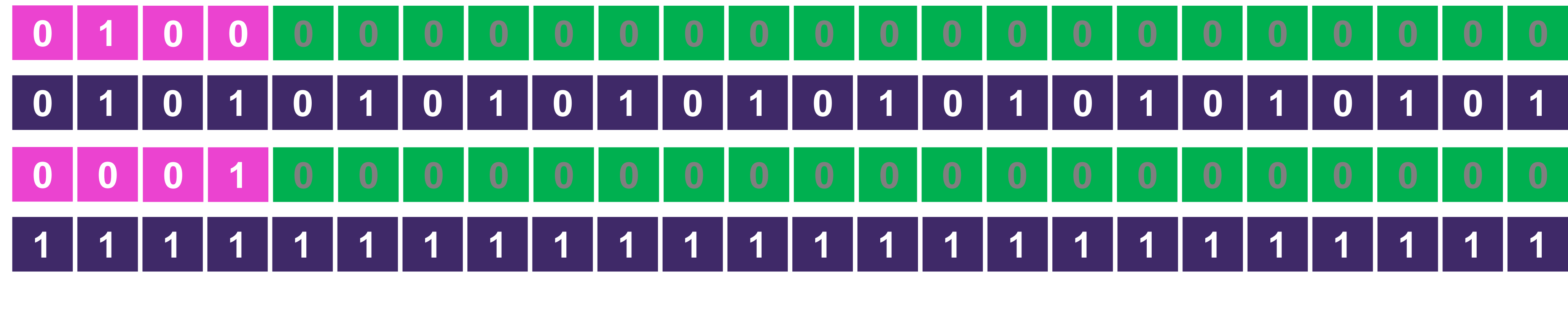
Recoding packets 101: Field Size GF(2^2)



- 01_b multiply 01_b equals 01_b (see look up table)
- 00_b multiply 11_b equals 00_b (see look up table)
- XOR both results
- 01_b XOR 00_b equals 01_b

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

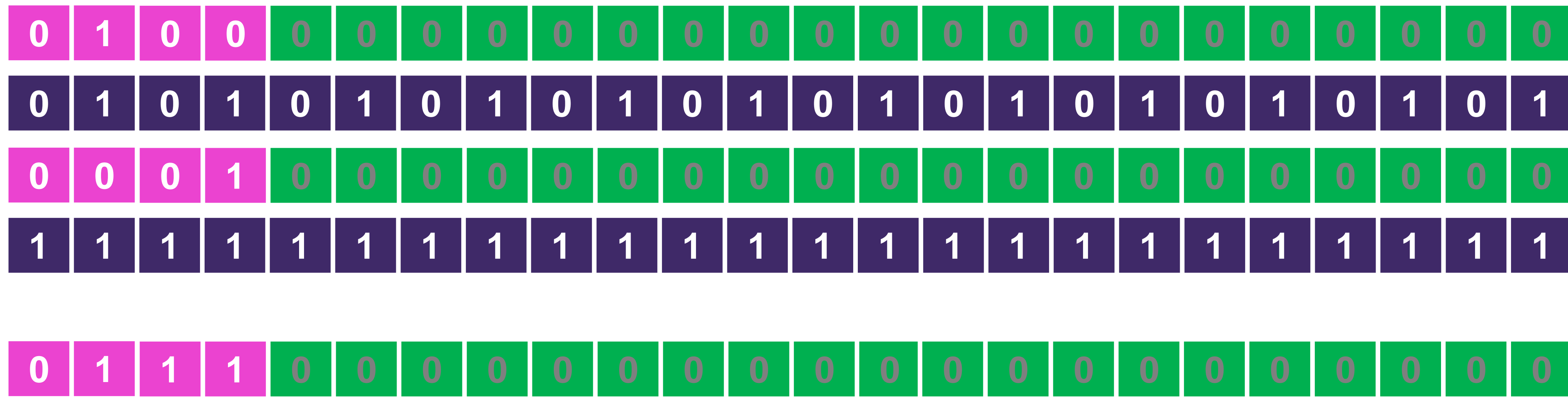
Recoding packets 101: Field Size GF(2^2)



- 00_b multiply 01_b equals 00_b (see look up table)
- 01_b multiply 11_b equals 11_b (see look up table)
- XOR both results
- 00_b XOR 11_b equals 11_b

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Recoding packets 101: Field Size GF(2^2)

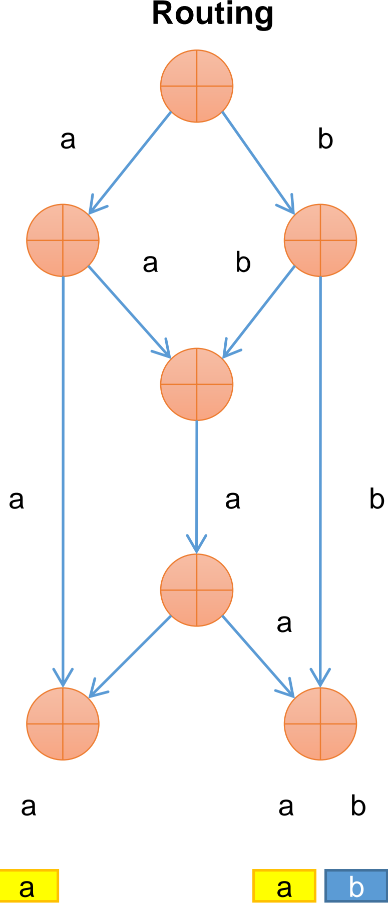


- Don’t forget the payload!

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

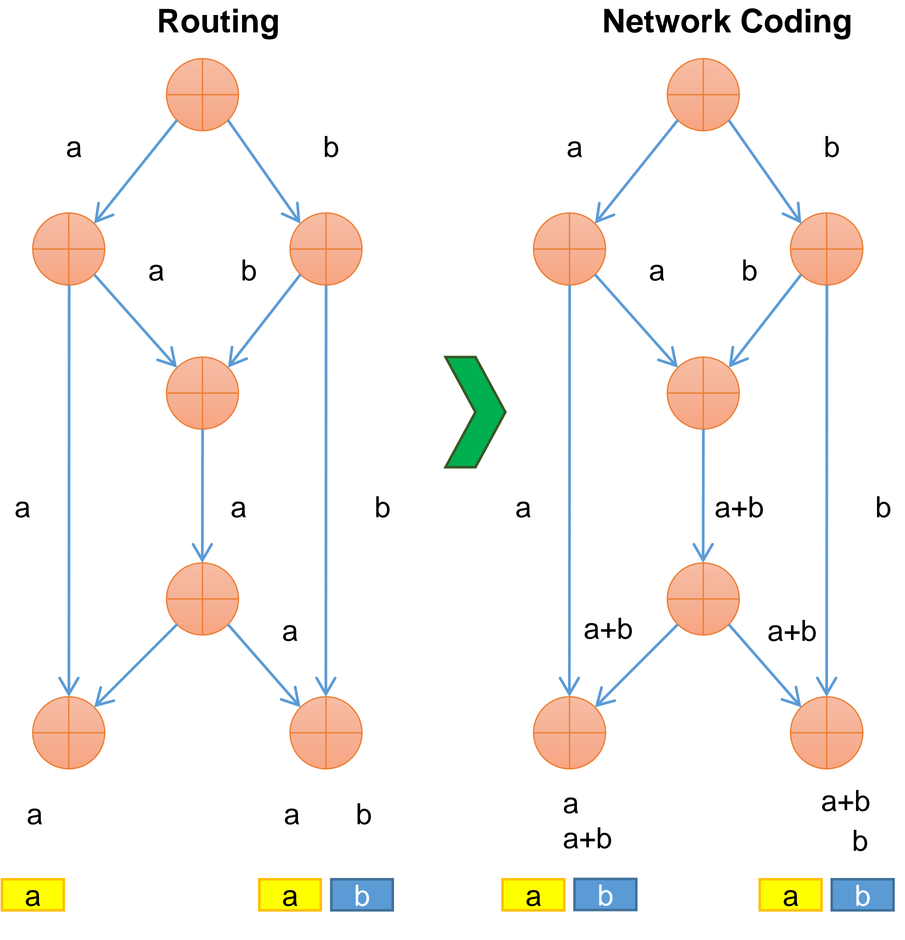
RLNC and the Butterfly

Network Coding



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

Network Coding



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

Routing

a b
a b
a a b
a
a a b
a | a b

Network Coding

a b
a b
a a+b b
a+b a+b
a a+b | a+b b
a b | a b

RLNC

GF(2^8)
3a+5b a+7b
3a+5b a+7b
GF(2^8)
3a+5b 76a+99b a+7b
76a+99b 76a+99b
3a+5b 76a+99b | a+7b 76a+99b
Decode GF(2^8)
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

What is the difference with respect to throughput and signaling?

Field Size Analysis

Ralf Kötter: „How bad is binary?“

Key Parameters of RLNC

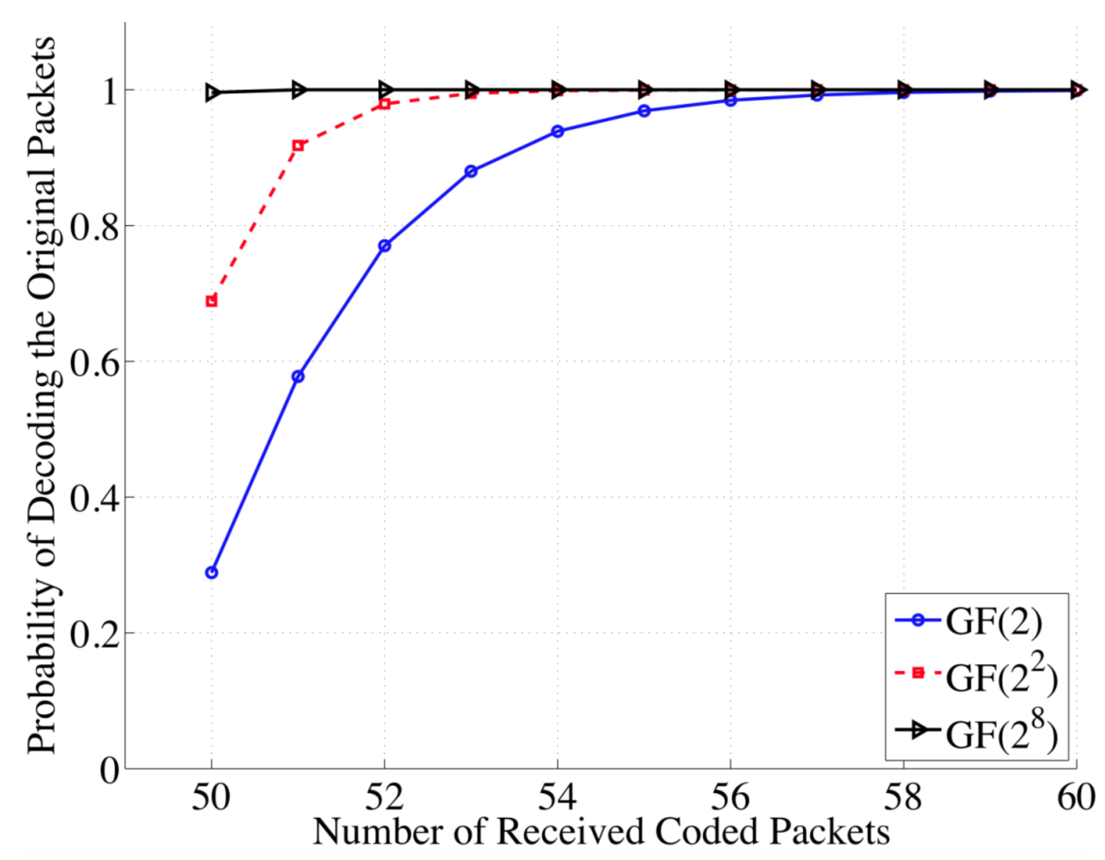
- Generation size: number of packets that are coded together.
- Field size: number of elements in the finite field

- Both have an impact on:
  - Performance in terms of linear dependency
  - Overhead in terms of encoding vector length or needed retransmissions due to linear dependency
  - Complexity

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 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 dependencies of the packets (doubles so to speak)



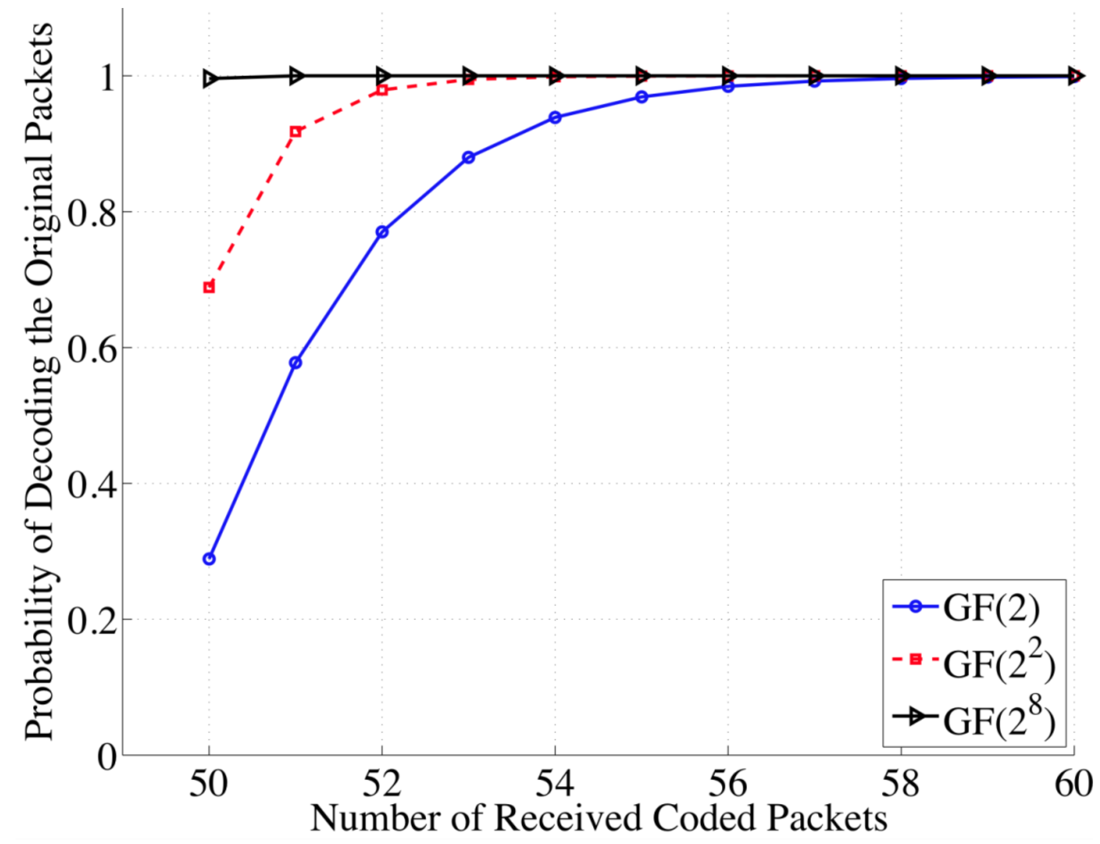
Field and Generation Size

Small field sizes are resulting in linear dependent coded packets.

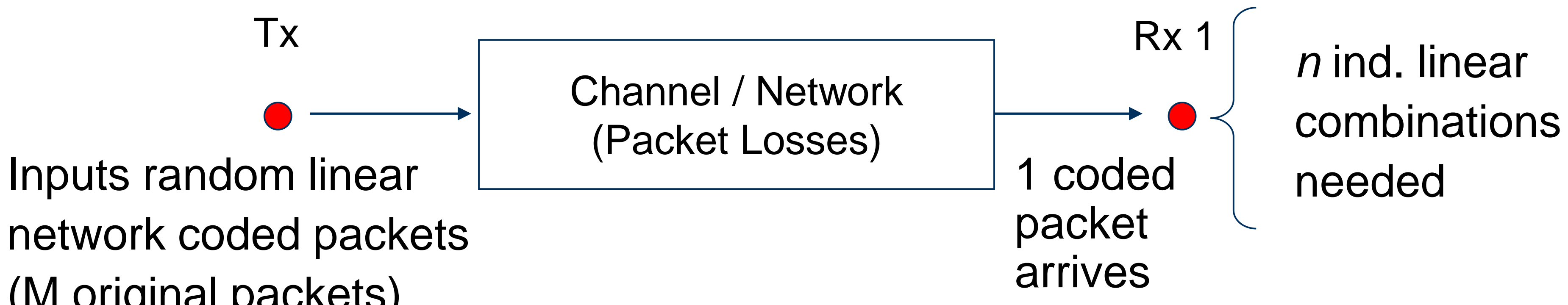
1.6 packets extra per generation in case of binary field sizes.

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

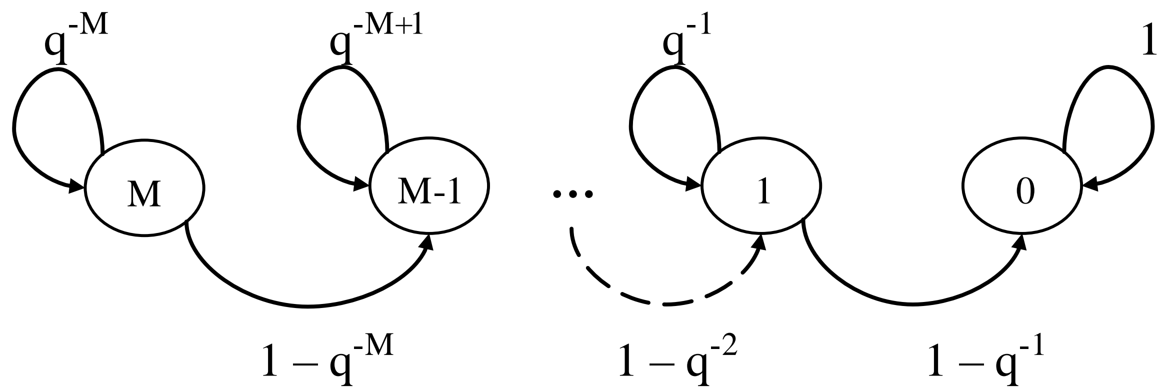
Theory is aiming for large field sizes and large generation sizes!



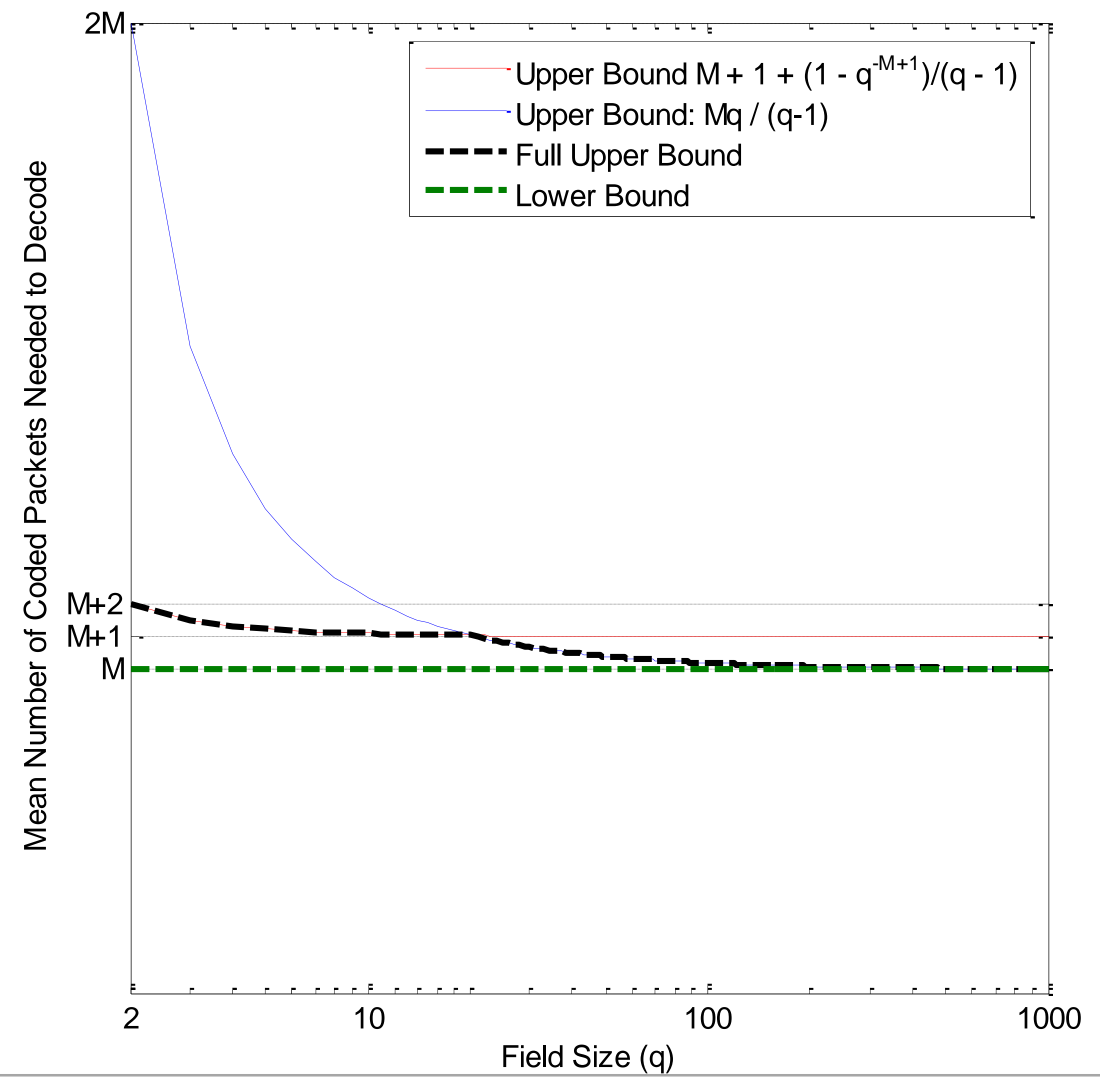
Field Size Analysis for RLNC

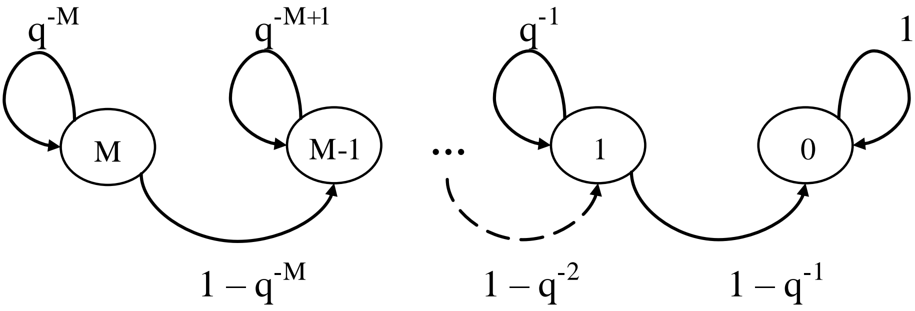


–Modeled as a Markov chain



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 and sufficiently large G equals

$$E_n = G + 1.6$$

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*	

q^{-M} q^{-M+1} q^{-1} 1

M M-1 … 1 0

$1 - q^{-M}$ $1 - q^{-2}$ $1 - q^{-1}$

Field Size Analysis

Linear dependency GF(2) G=16

degree of freedom (DOF)

15
13
11
9
7
5
3
1

0 0,2 0,4 0,6 0,8 1 1,2

extra packets per DOF

Field Size Analysis

Linear dependency GF(2) G=16

degree of freedom (DOF)

15
13
11
9
7
5
3
1

1.6

0 | 0,2 | 0,4 | 0,6 | 0,8 | 1 | 1,2

extra packets per DOF

Field Size Analysis

Overhead in dependency of generation size

extra linear dependent packets per generation

1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

generation

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
```

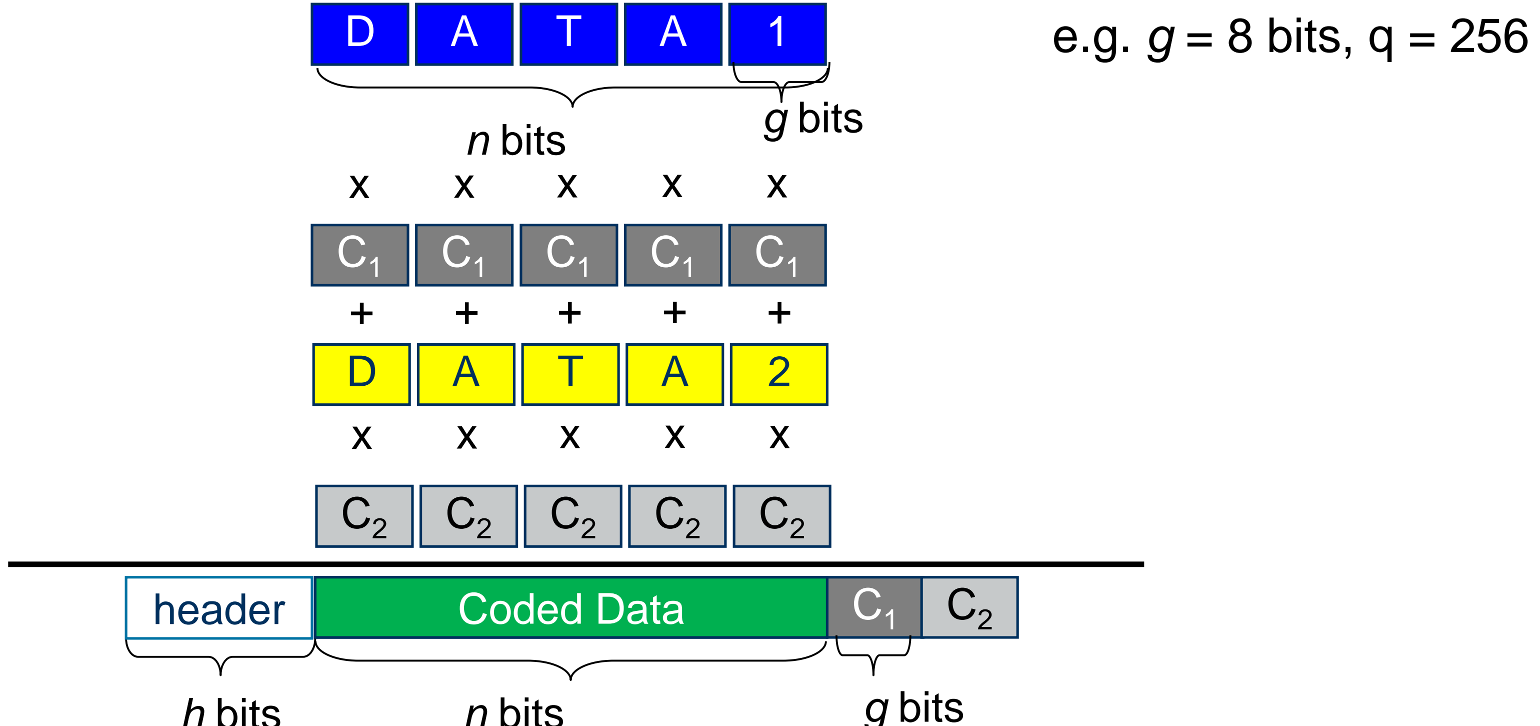
It is only 1.6 extra packets!

Recoding Potential

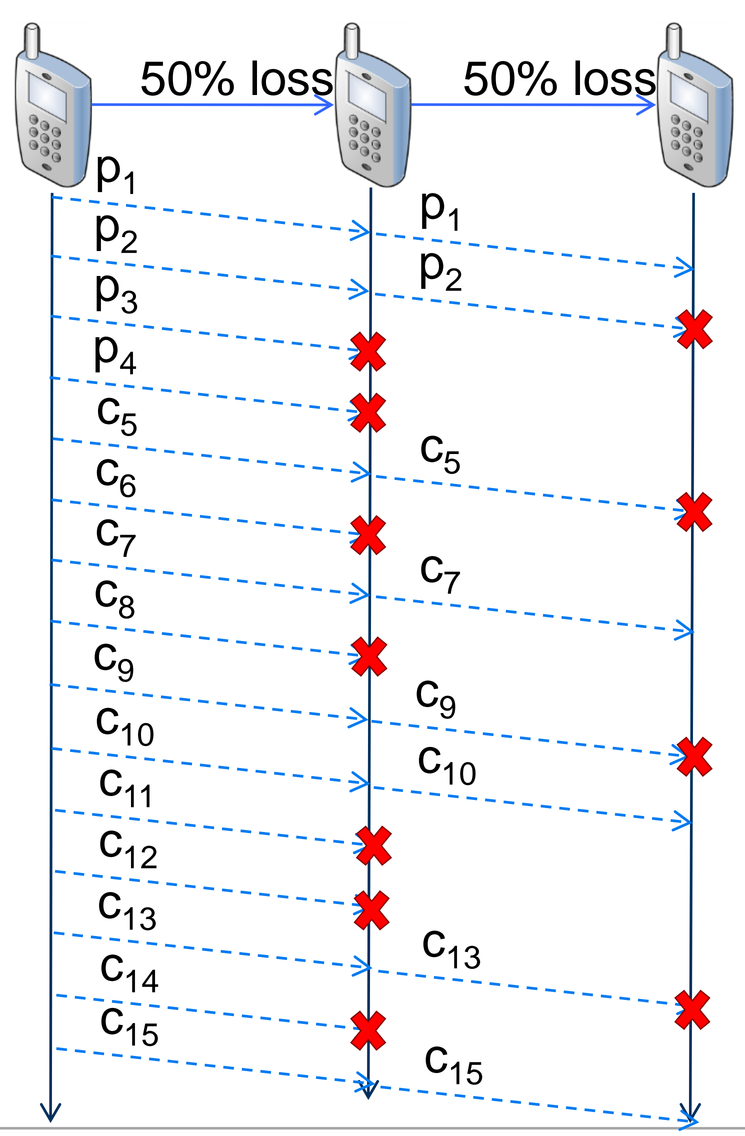
Generating a Coded Packet

- Generating a linear network coded packet (CP)
- Operations over finite field of size.

$$CP_j = \sum_i C_i P_i$$



No Recoding

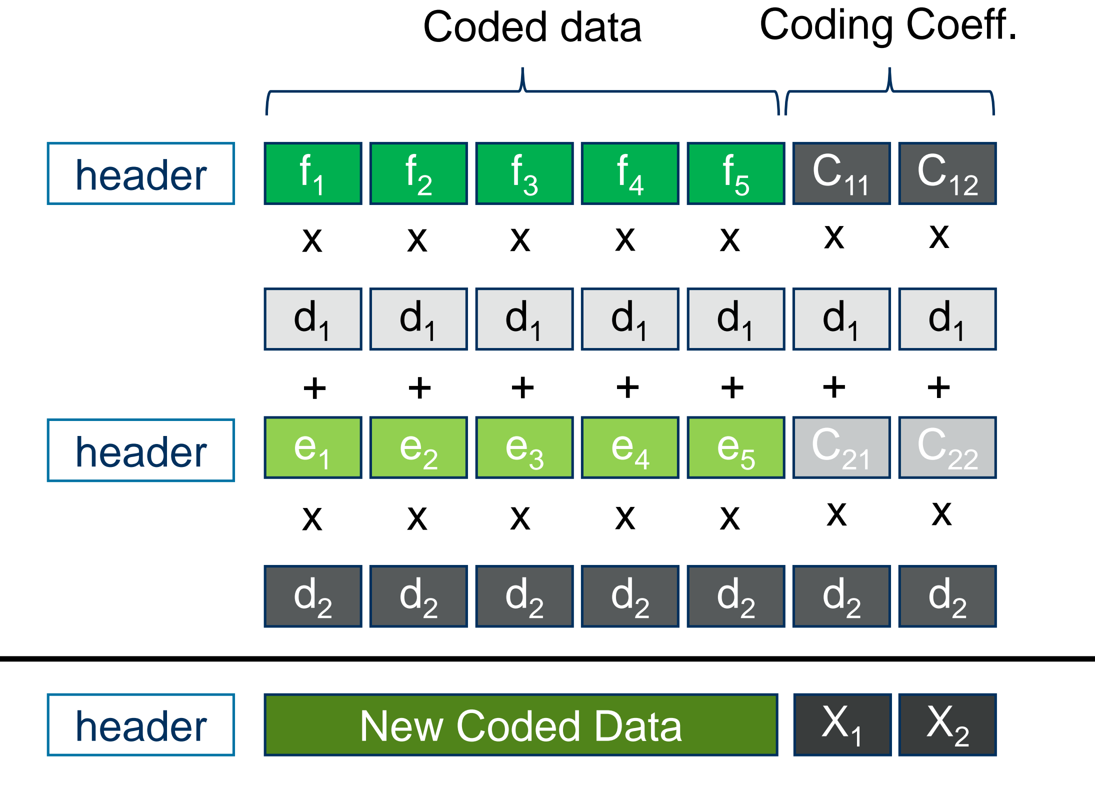


- Simple operation: forward
- Structure of code is preserved
- Issues
  - Delay per batch of packets
  - Missing transmission opportunities
  - Equivalent loss probability: compounding each channel's loss

- 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)

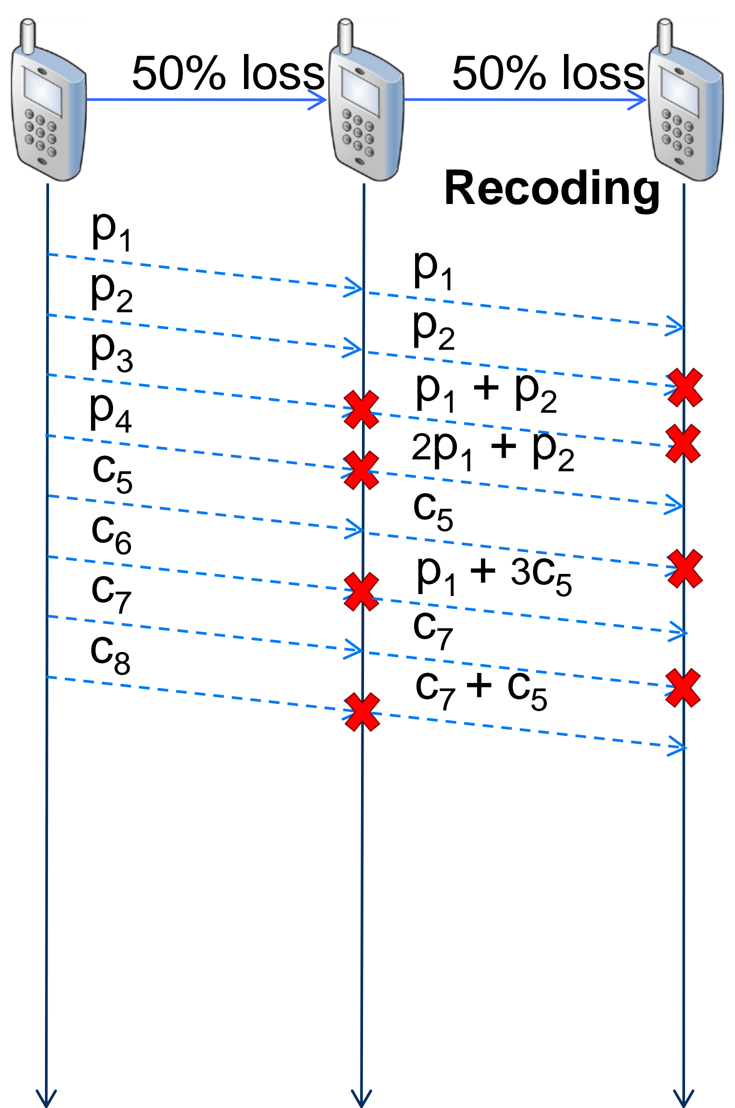


$X_1 = d_1\,C_{11} + d_2\,C_{21}$ and $X_2 = d_1\,C_{12} + d_2\,C_{22}$

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

Thus, $d_1 f + d_2 e = d_1C_{11}P_1 + d_1C_{12}\,P_2 + d_2C_{21}P_1 + d_2C_{22}\,P_2 = X_1P_1 + X_2P_2$

Recoding Packets

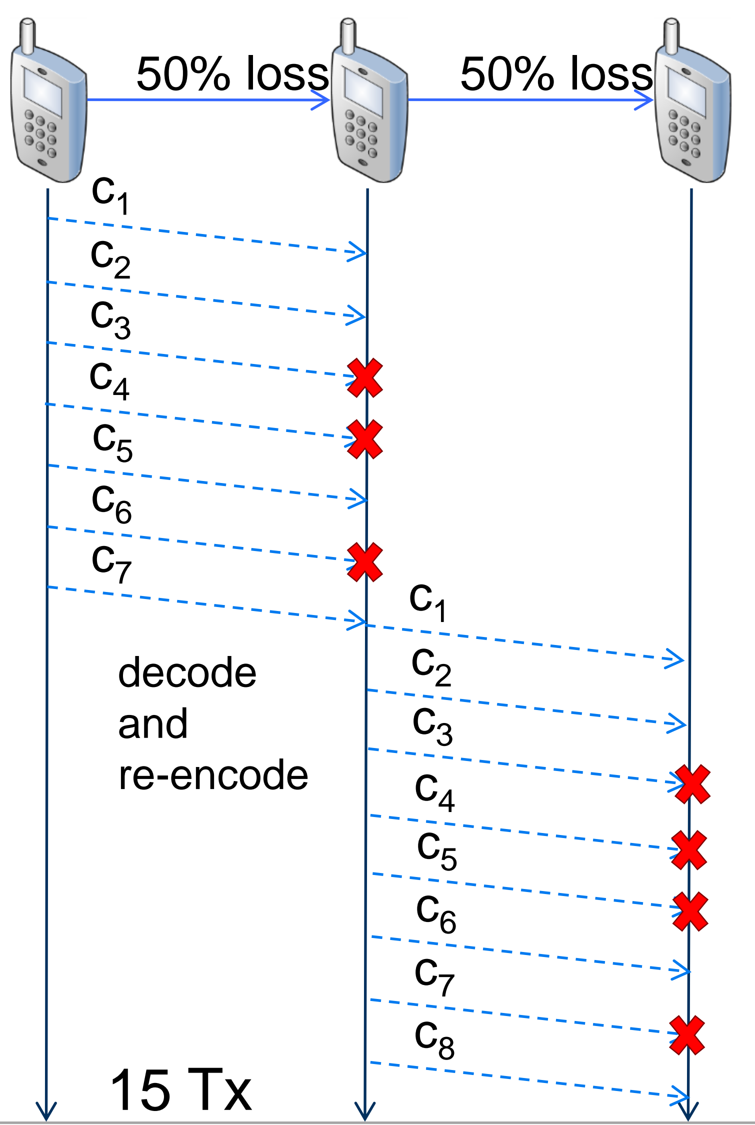


15 Tx

- 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*:

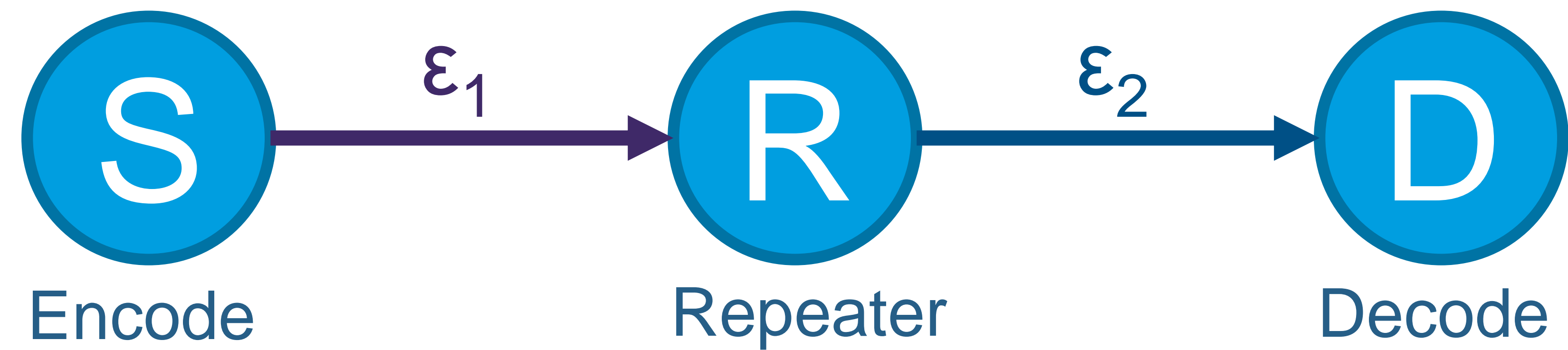
$$\min\{1-e_1, 1-e_2\}$$

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

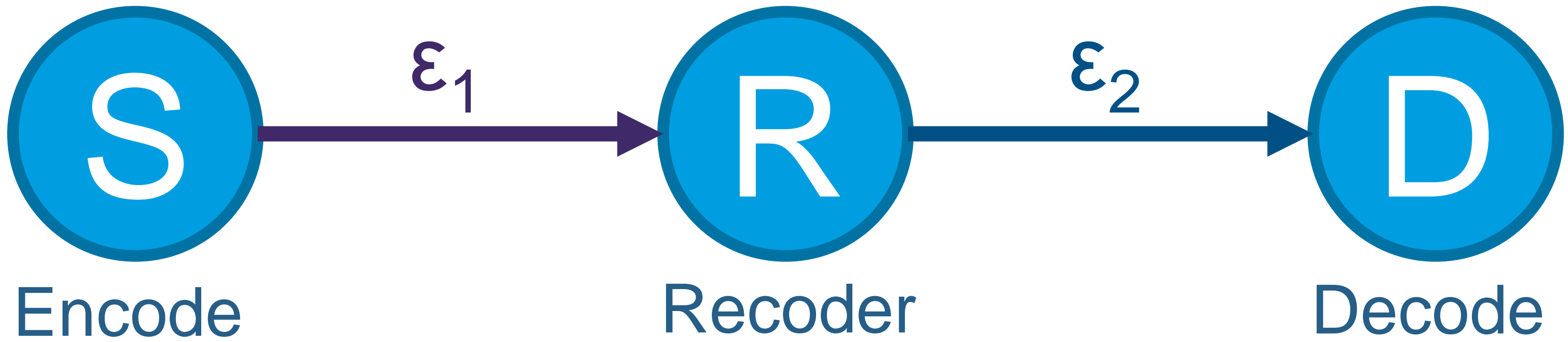


Reed Solomon
LDPC
LT
Raptor

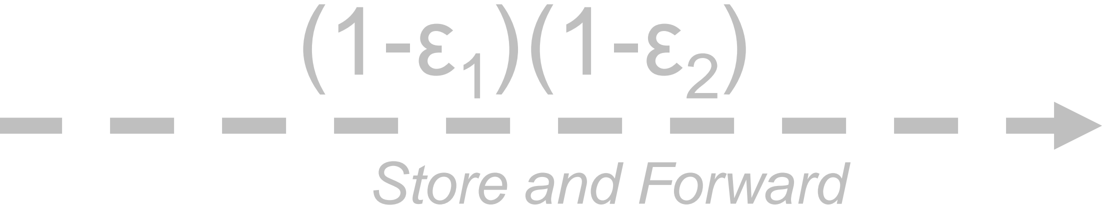
$(1-\varepsilon_1)(1-\varepsilon_2)$

Store and Forward

Network Coding



Reed Solomon
LDPC
LT
Raptor



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

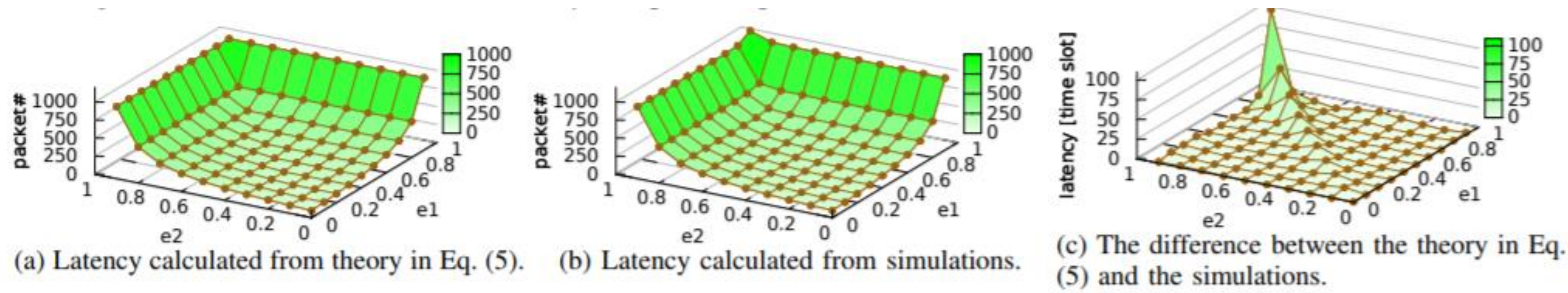
With recoding

In [4]:
```python
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

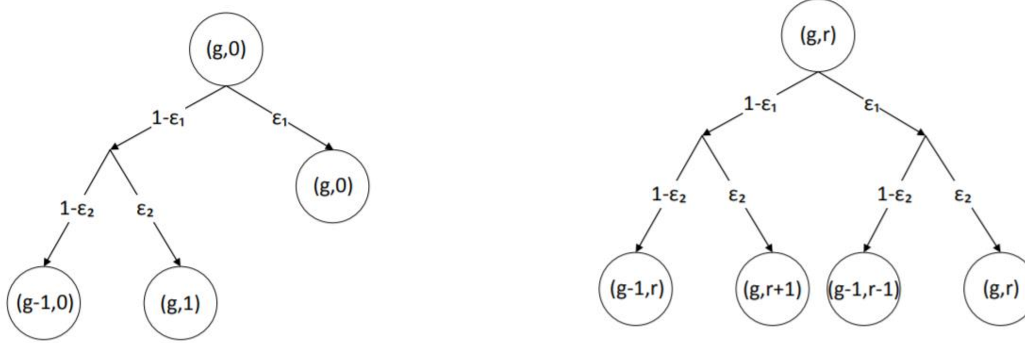
- If error rate is high and the same (or close) for both links, the theory and the practical implementation do not match.
- Underlines the importance for implementation and theory → Theory that matters!



(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.

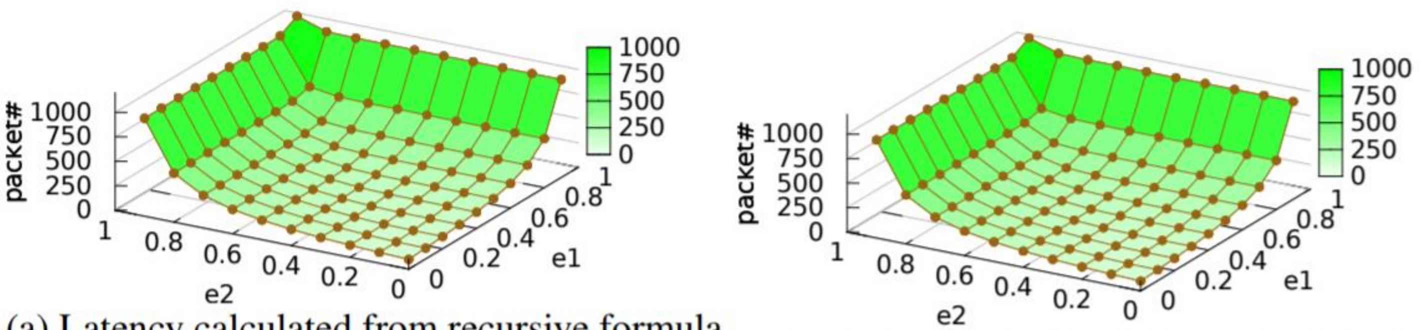
Sharkfin Problem

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



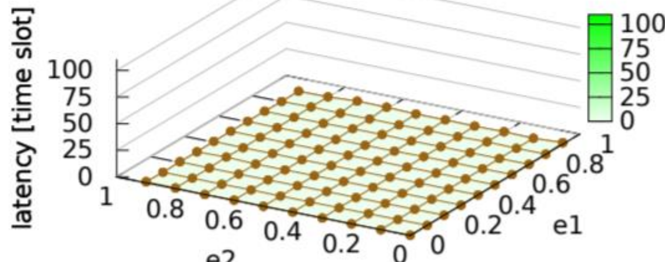
(a) State transition graph when the recoder is empty.

(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

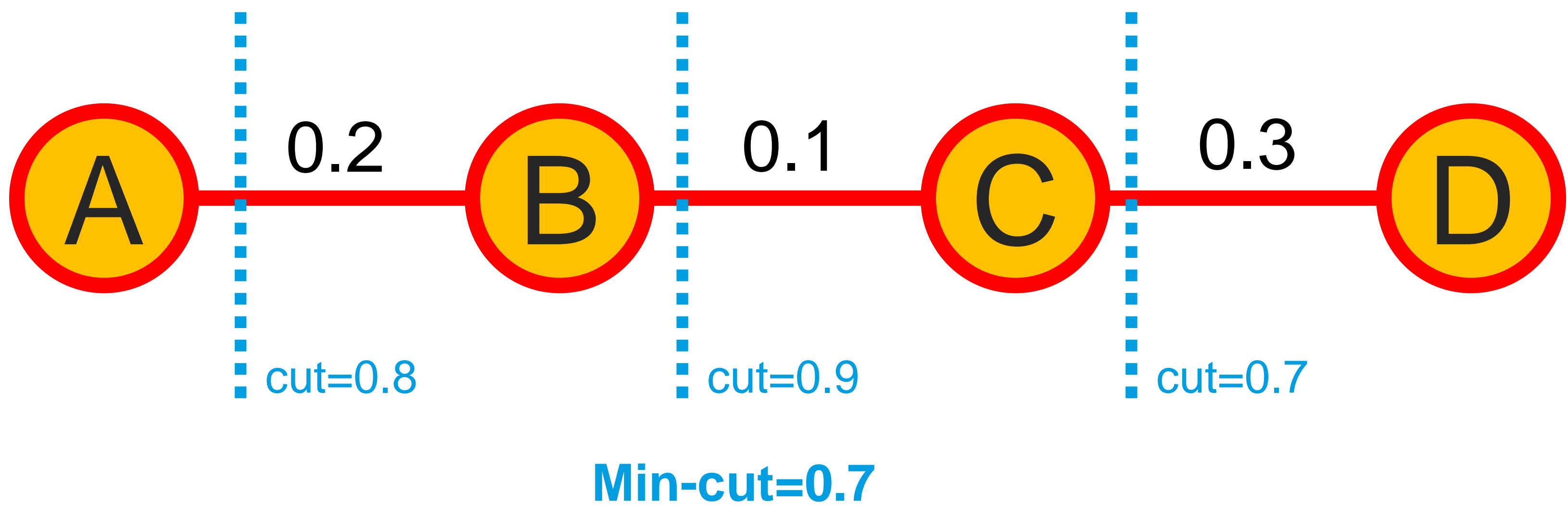
Optimal and Dynamic Loss Compensation

- Uncoded ~50%
- Coded
  - Hop by hop: Efficient but creates delay
  - E2E (A → D) plan for the worst link (30%)
  - RLNC: Recoding (as efficient as hop by hop without delay)

The Recoding Advantage

Optimal and Dynamic Loss Compensation

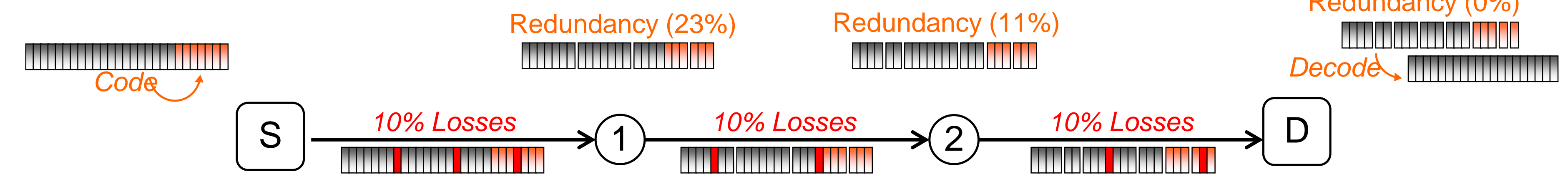
- Uncoded ~50%
- Coded
  - Hop by hop: Efficient but creates delay
  - E2E (A → D) plan for the worst link (30%)
  - RLNC: Recoding (as efficient as hop by hop without delay)



The Recoding Advantage

Optimal and Dynamic Loss Compensation

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



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

Required Redundancy: 11%

Code

Redundancy (0% → 11%)

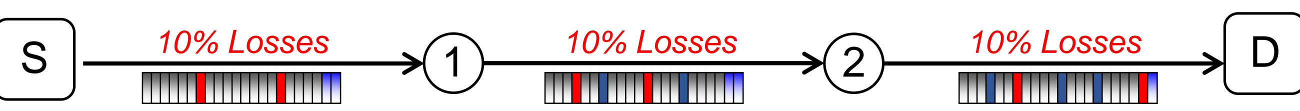
Recode

Redundancy (0% → 11%)

Recode

Redundancy (0%)

Decode



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

Optimal and Dynamic Loss Compensation

A 0.1 A 0.1 A

0.1 0.1

S 0.1 B 0.1 B 0.1 D

0.1 0.1

C 0.1 C 0.1 C

The Recoding Advantage

Optimal and Dynamic Loss Compensation

A A A
0.1 0.1
0.1 0.1
S 0.1 B 0.1 B 0.1 D
0.1 0.1
C C C
0.1 0.1

Min-cut=2.7

The Recoding Advantage

Optimal and Dynamic Loss Compensation

RLNC provides multi-path communication throughput of 2.7 without complex scheduling at the source

The Recoding Advantage

Optimal and Dynamic Loss Compensation

The Recoding Advantage

Optimal and Dynamic Loss Compensation

A A A
0.1 0.4
0.2 0.1
S B B D
0.1 0.2 0.3
0.4 0.4
C C C
0.3 0.4

Min-cut=1.9

The Recoding Advantage

Optimal and Dynamic Loss Compensation

Tip: Sharkfin

S, A, A, A, B, B, C, C, C, D

0.2, 0.1, 0.4, 0.1

0.1, 0.2, 0.3

0.3, 0.4

RLNC provides multi-path communication throughput of 1.9 without complex scheduling at the source

The Recoding Advantage plus Multipath Advantage

Native Bandwidth Aggregation

Redundancy (5.3%) Recode (0% → 25%) Recode (0% → 11%) Redundancy (0%)

S —5% Losses→ 1 —20% Losses→ 2 —10% Losses→ D

Path 1

Redundancy (11%) Recode (0% → 11%) Recode (0% → 11%) Redundancy (0%)

code & distribute

S —10% Losses→ 3 —10% Losses→ 4 —10% Losses→ D

combine & decode

Path 2

Redundancy (11%) Recode (0% → 11%) Recode (0% → 11%) Redundancy (0%)

S —10% Losses→ 5 —10% Losses→ 6 —10% Losses→ D

Path 3

Summing without scheduling

multipath

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

Other Recoding Issues

25% loss
25% loss
50% loss

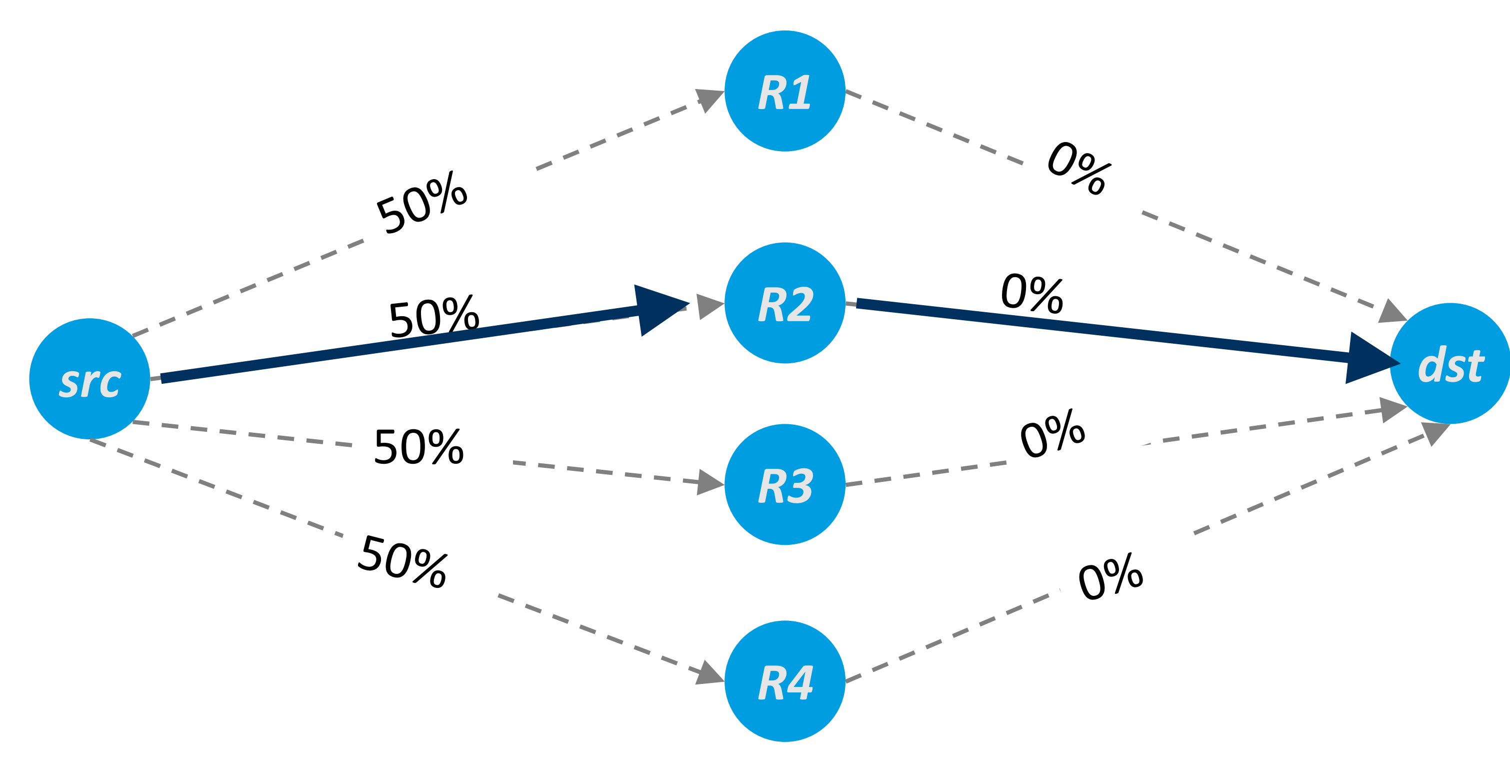
p_1
p_1
p_2
p_2
p_3
p_1? p_2? $p_1 + p_2$?
p_4
p_4

25% loss
25% loss
50% loss

p_1
p_2
$p_1 + p_2$
p_3
p_4
$p_2 + p_4$

Other Recoding Issues

Best single path → Prob. of loss 50%



Other Recoding Issues

Best single path → Prob. of loss 50%

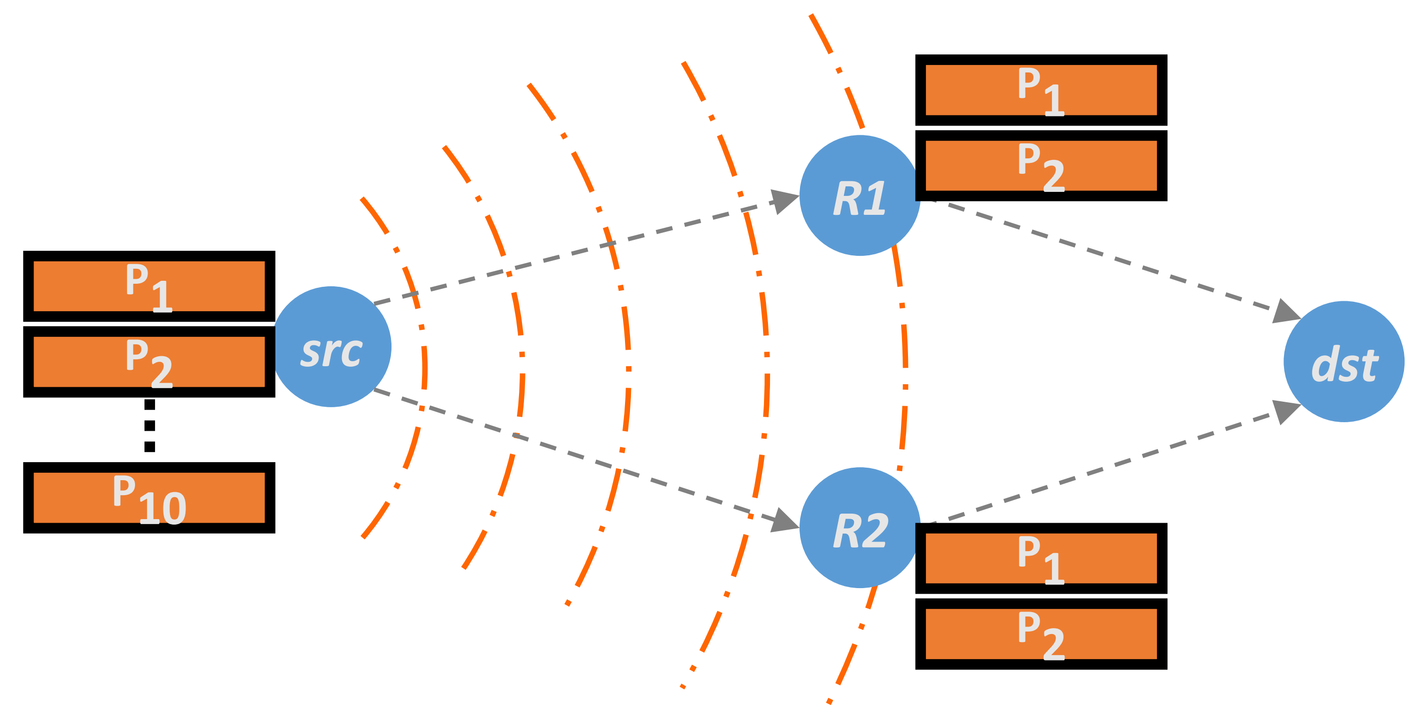
src
R1
R2
R3
R4
dst
50%
50%
50%
50%
0%
0%
0%
0%

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

Overlap in received packets → Routers forward duplicates

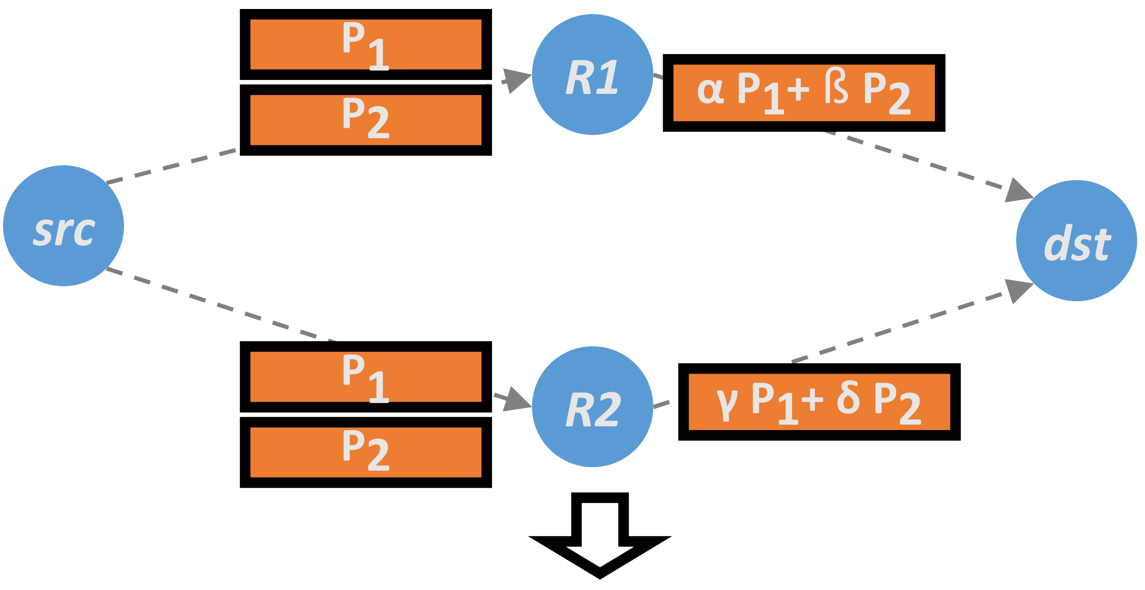
Challenge with Using Spatial Diversity

Overlap in received packets → Routers forward duplicates

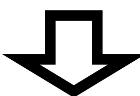


Challenge with Using Spatial Diversity

Each router forwards random combinations of packets

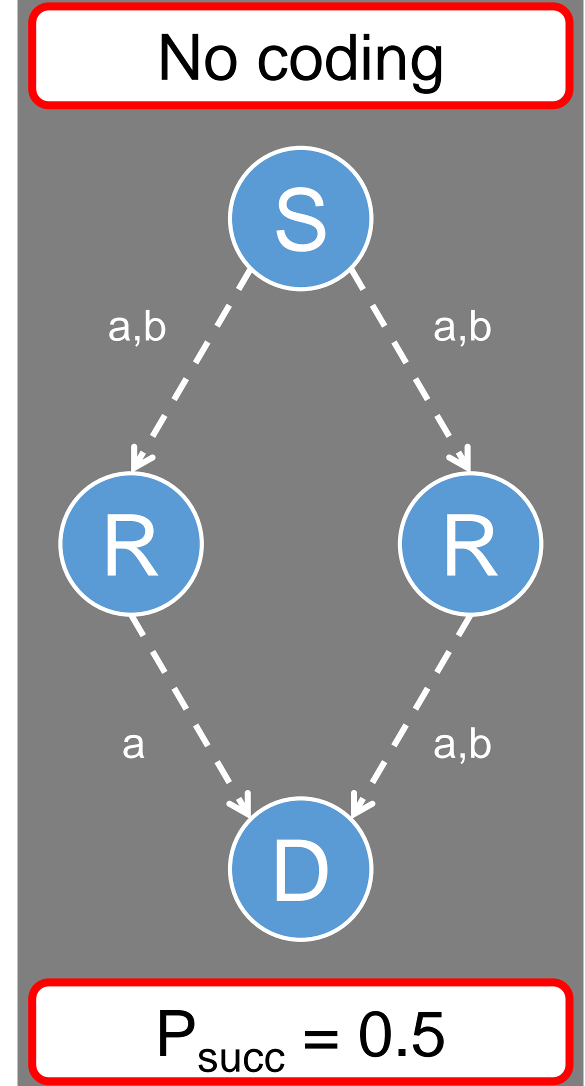


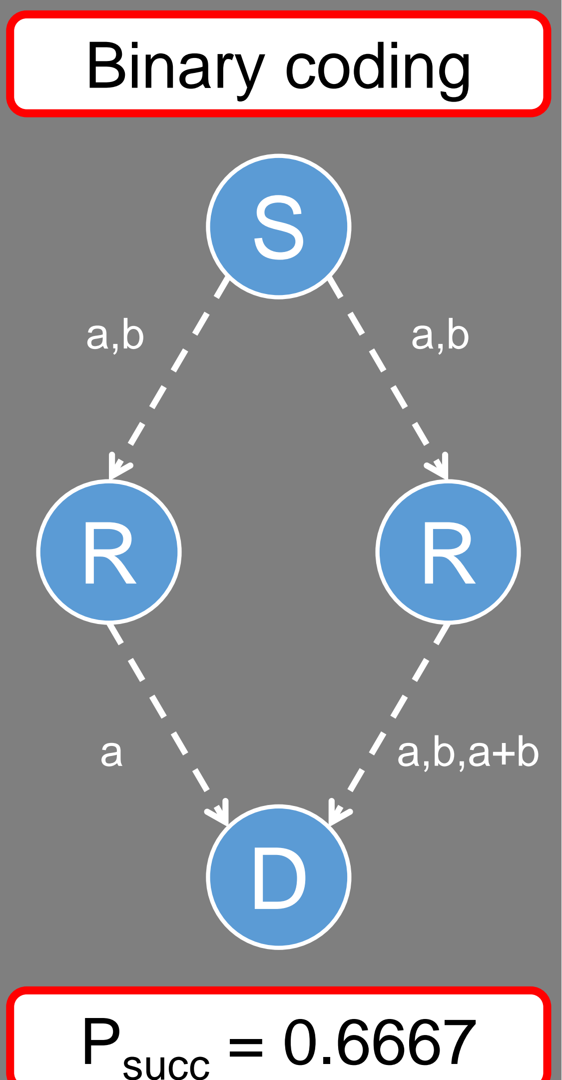
Randomness prevents duplicates

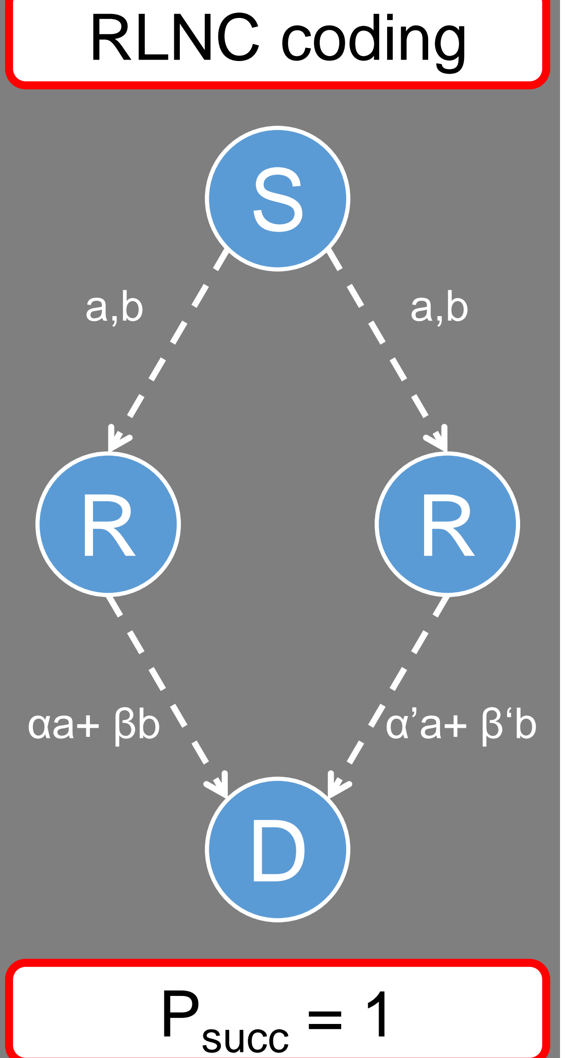


Network coding exploits spatial diversity to improve dead spots

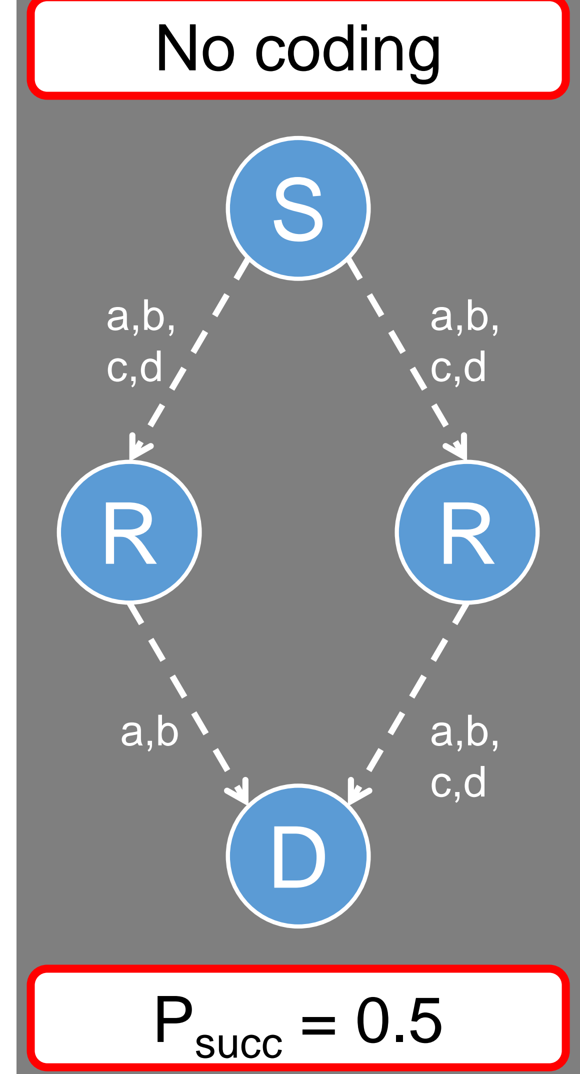
Impact of Recoding

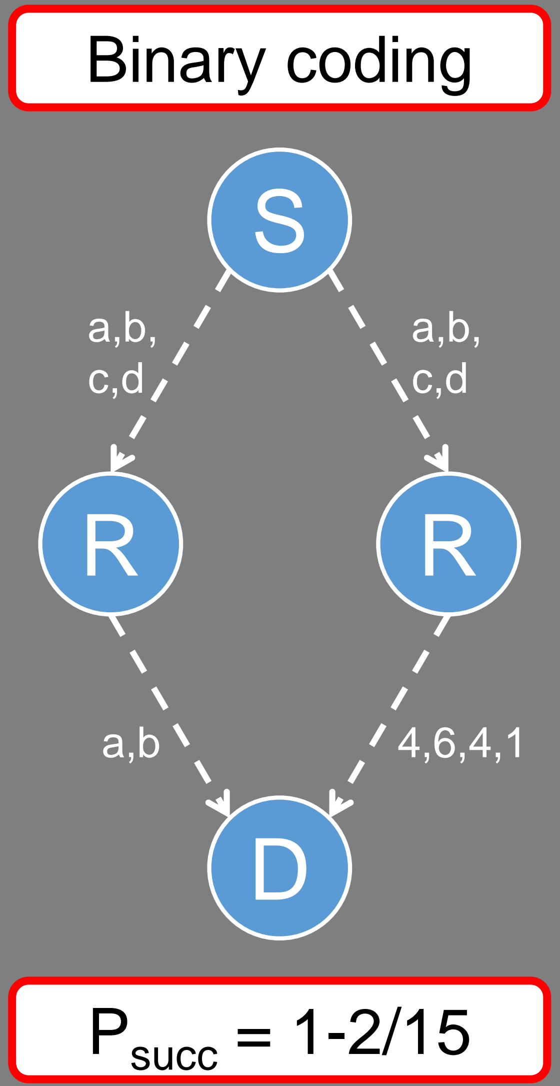


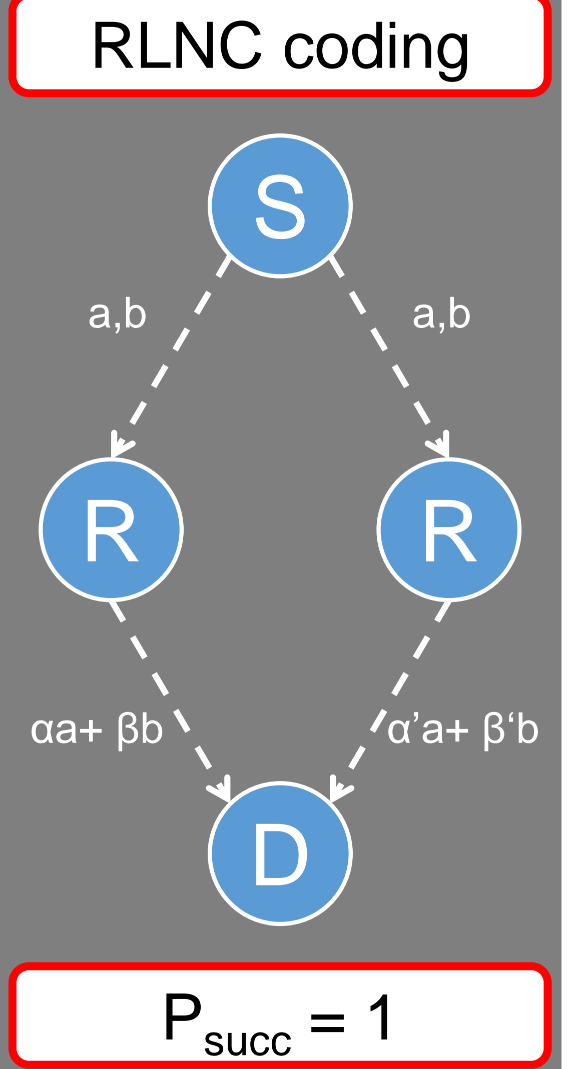




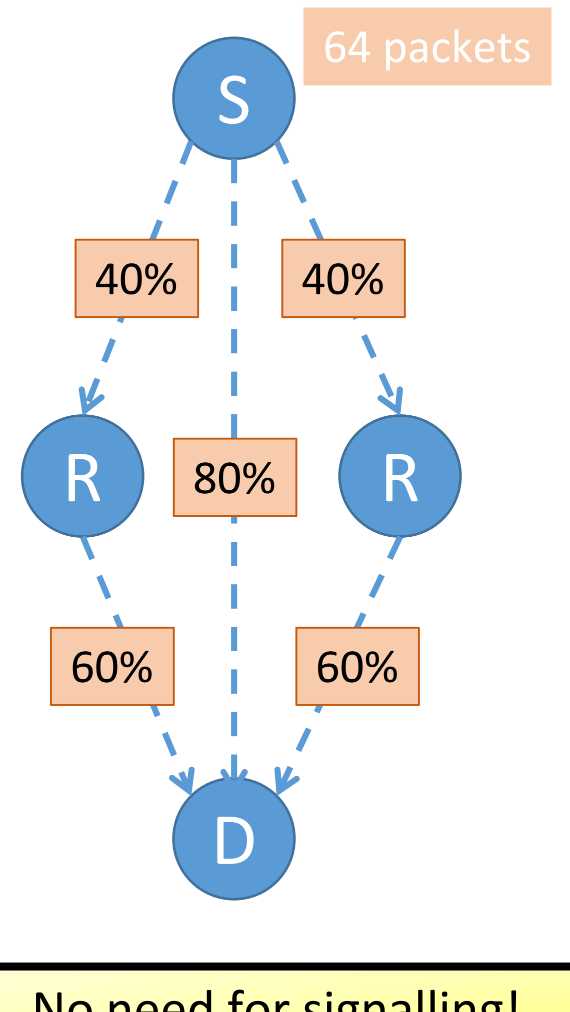
Impact of Recoding

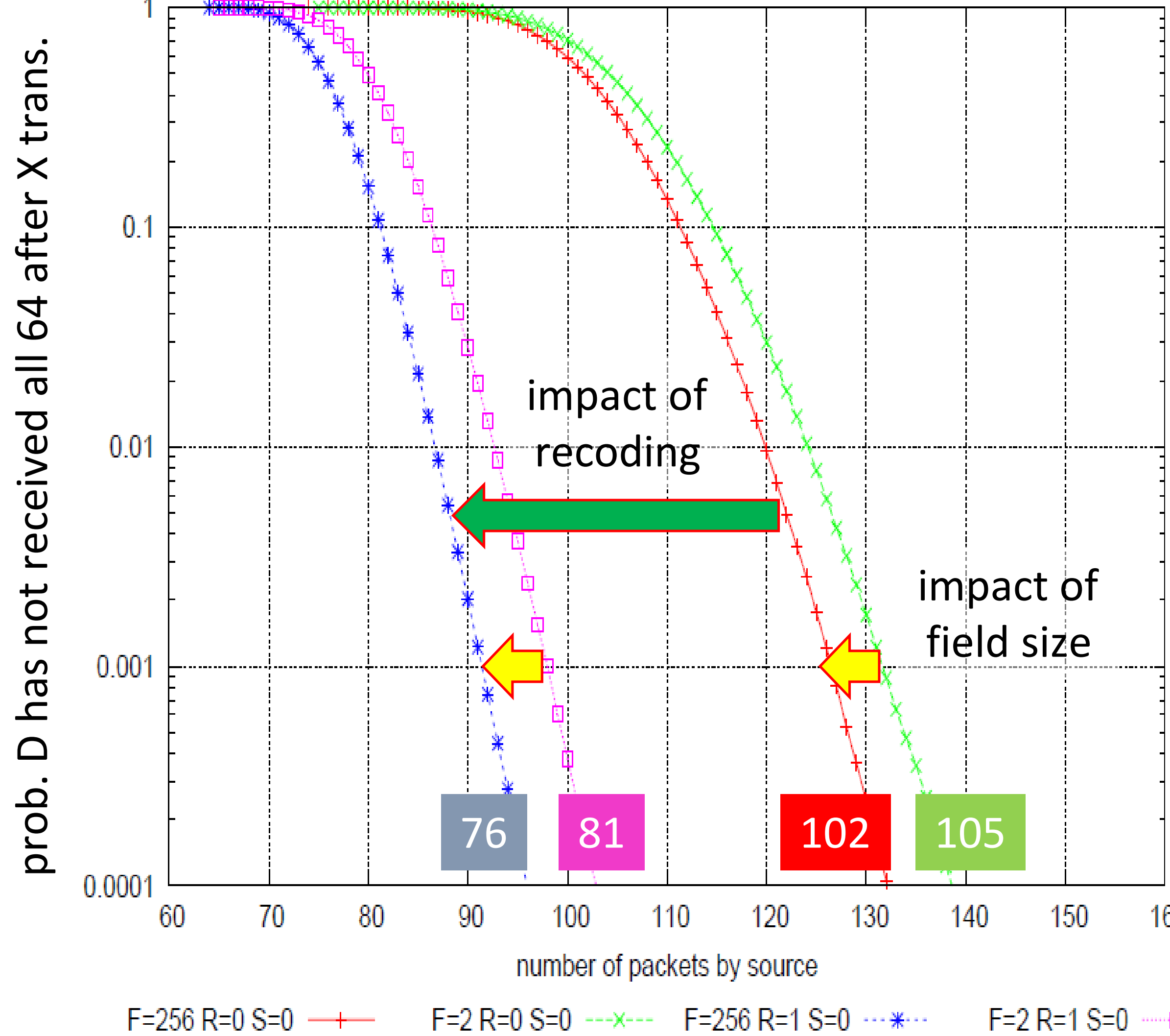






Impact of Recoding

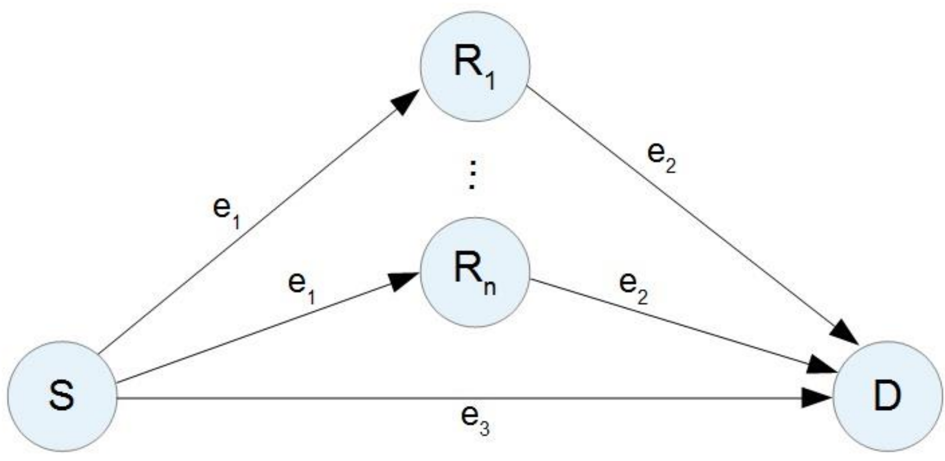




No need for signalling!

http://notebook.deutsche-telekom.rocks

- Kodo_Multipath_Example.ipynb example

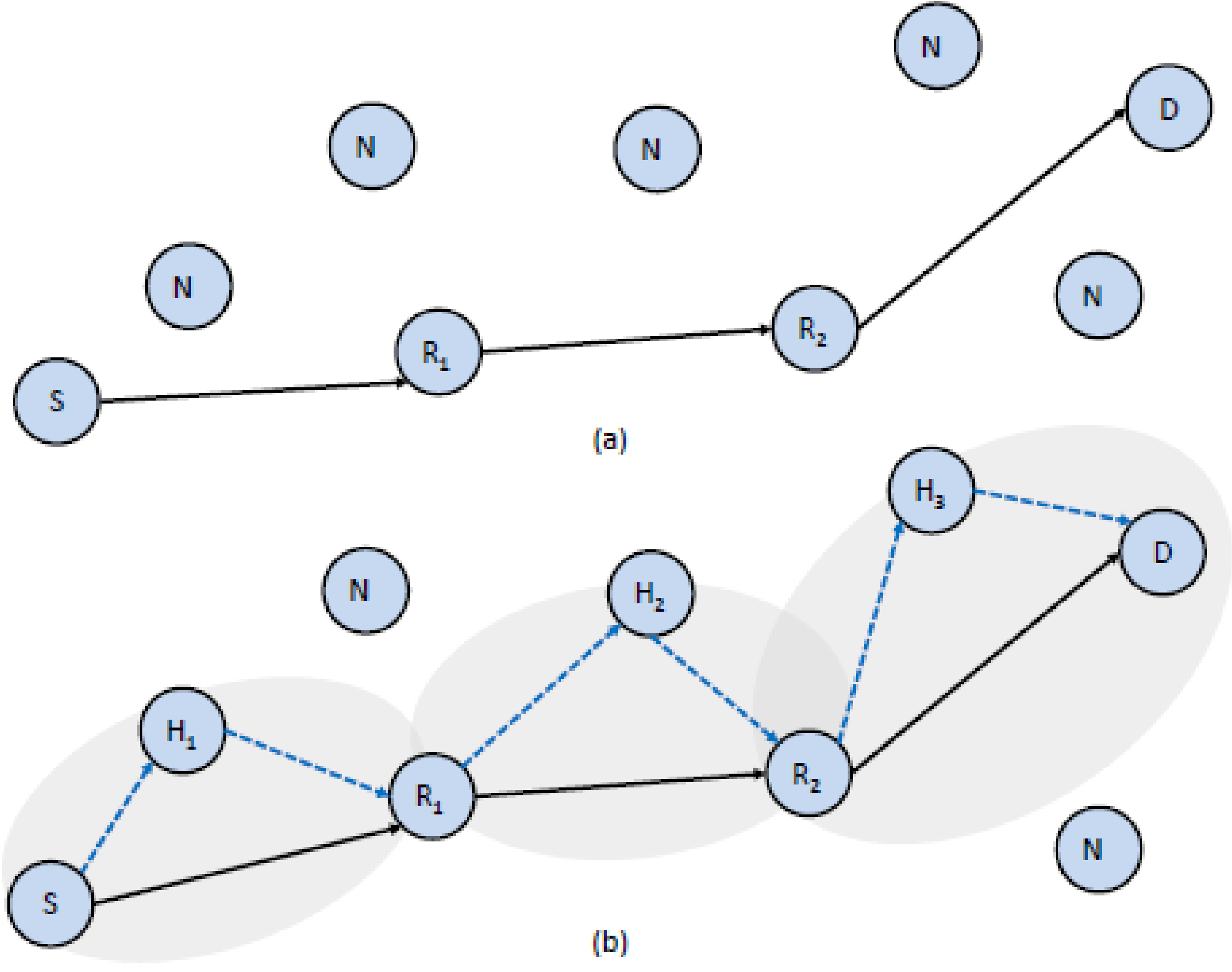


#Relays	1	2	3	4	5	6	7	8	9
Repeater									
Recoder									
Gain									

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

Whenever I get a slot to transmit I will send something recoded!

I wait and play cool until I am sure I am saying something meaningful!

Whenever I receive a new packet, I will recode it!

H_{i+1}

$e_1^{(i+1)}$

$e_2^{(i+1)}$

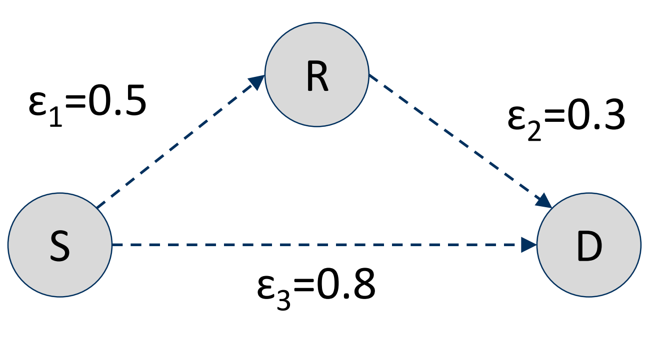
R_i

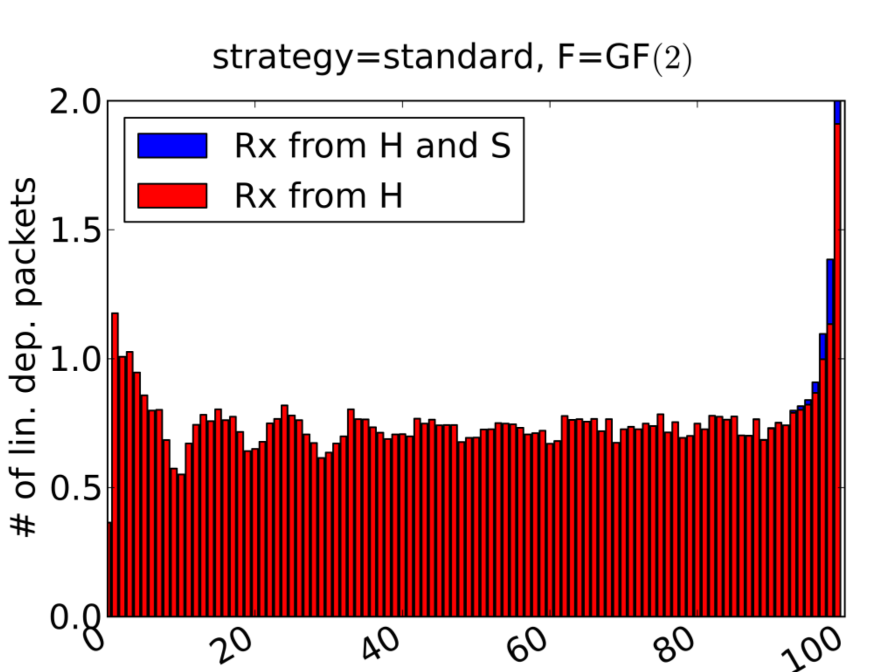
R_{i+1}

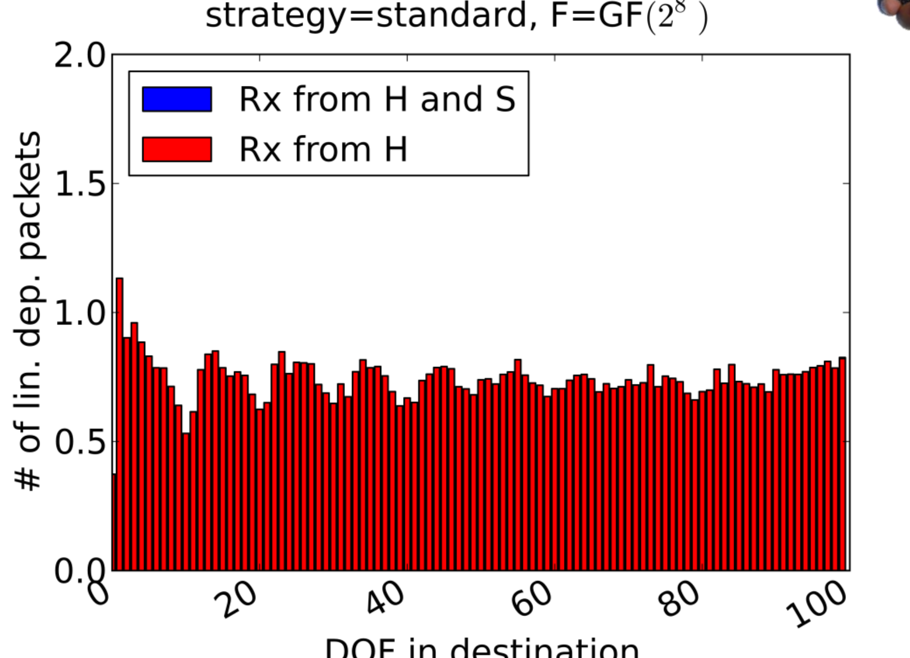
$e_3^{(i+1)}$

RLNC in real meshed

Agnostic recoding

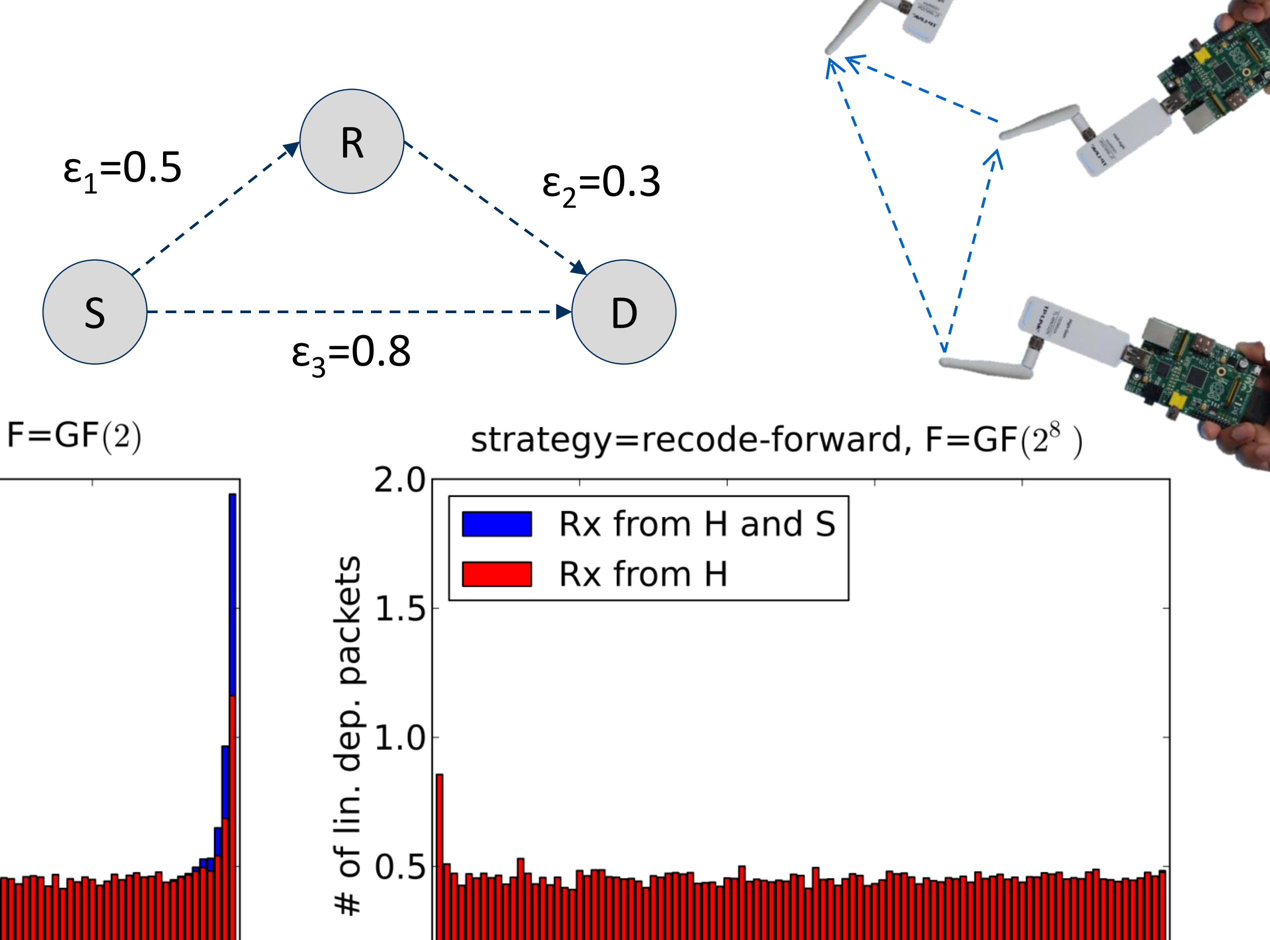


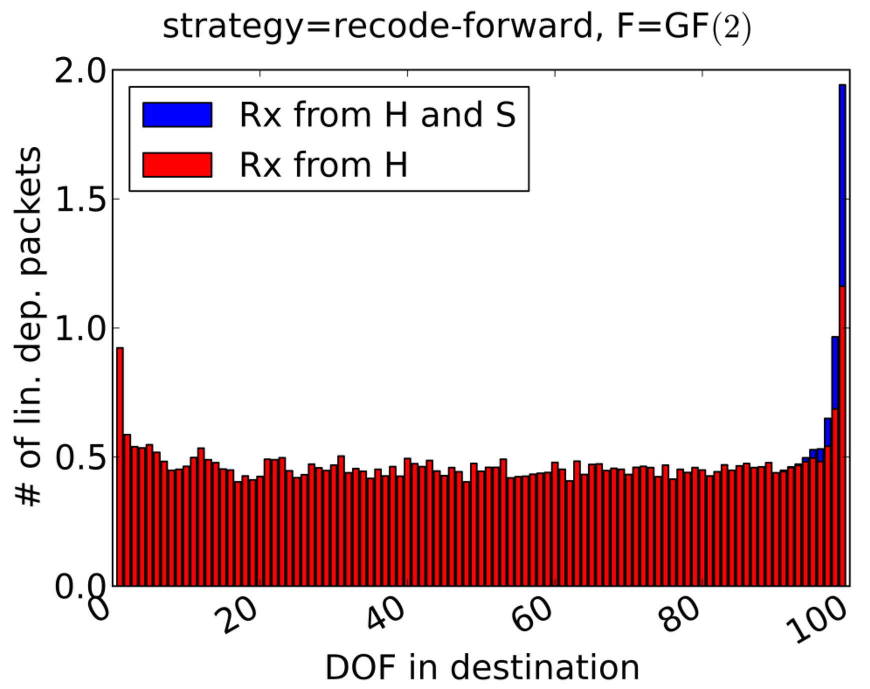




RLNC in real meshed

Rate-sensitive recoding





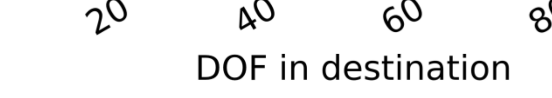
strategy=recode-forward, F=GF(2^8)

Rx from H and S

Rx from H

# of lin. dep. packets

DOF in destination



RLNC in real meshed

PlayN Cool

S, R, D
$\varepsilon_1=0.5$
$\varepsilon_2=0.3$
$\varepsilon_3=0.8$

strategy=PlayNCool, F=GF(2)
Rx from H and S
Rx from H
only 3% overhead
\# of lin. dep. packets
DOF in destination

strategy=PlayNCool, F=GF(2^8)
Rx from H and S
Rx from H
only 1% overhead
\# of lin. dep. packets
DOF in destination

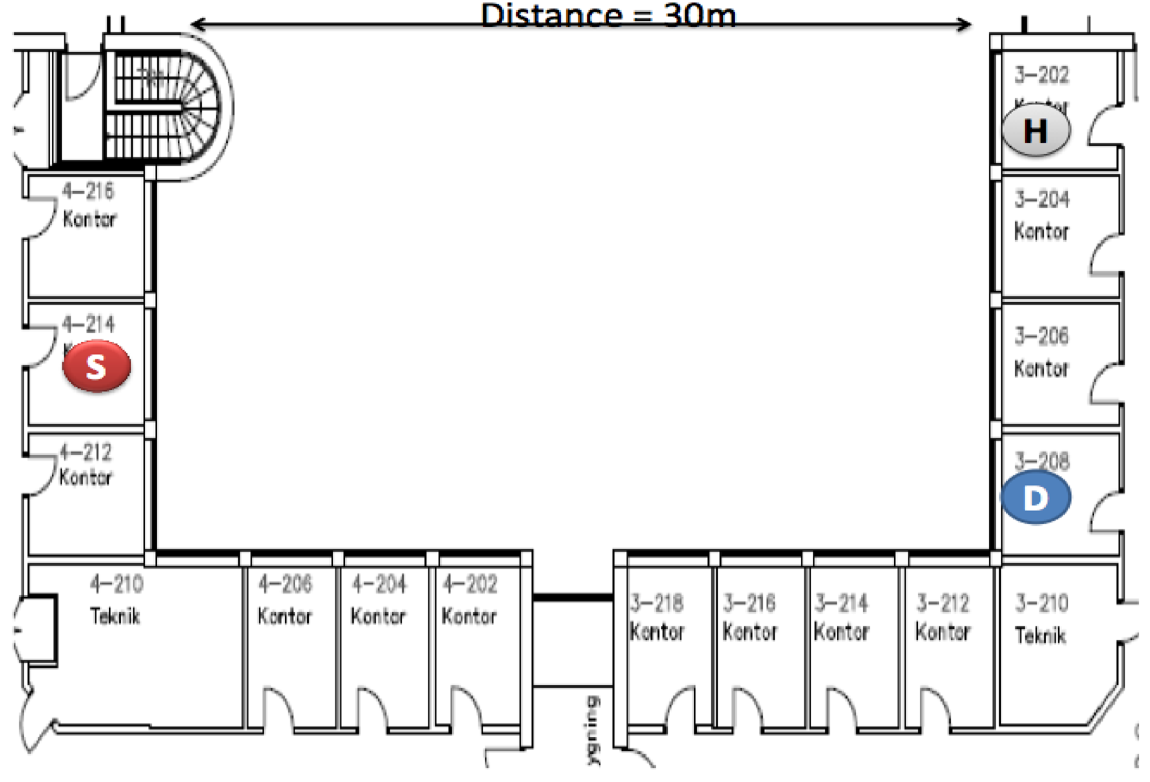
RLNC in real meshed

PlayN Cool

- Link quality information is inaccurate and not stable over run time
- Link quality information is not always available
- Retrieving the link quality information increases the overhead
- The waiting time (W), time the recoder waits to increase knowledge, is adaptive based on a feedback packet from the destination node.
- Feedback shows the usefulness of the packet
  - W decreases by UPDATE_VALUE if packet is useful
  - W increases by UPDATE_VALUE if RX packet is useless

RLNC in real meshed

PlayN Cool



The testbed deployed at AAU

Raspberry Devices

RLNC in real meshed

PlayN Cool

Avg. # linear dependent packets

The number of training generations

The effect of number of training generation

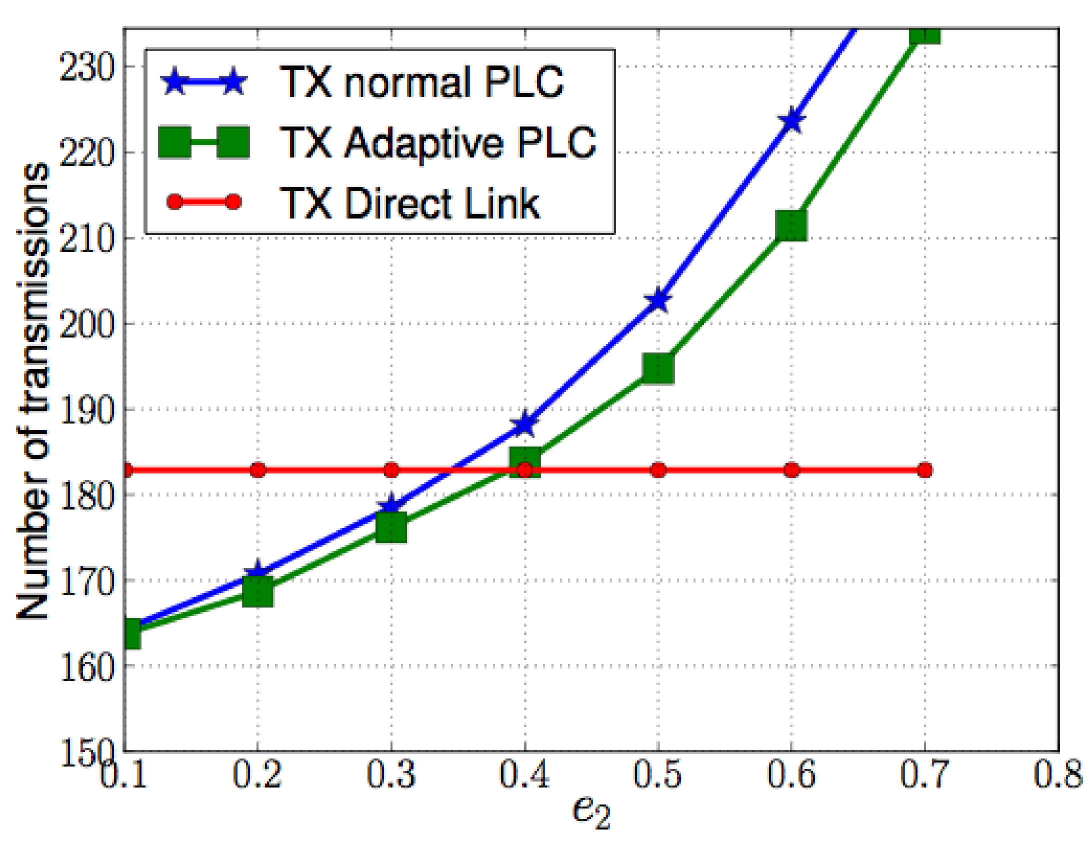
Total number of transmissions

Updating value

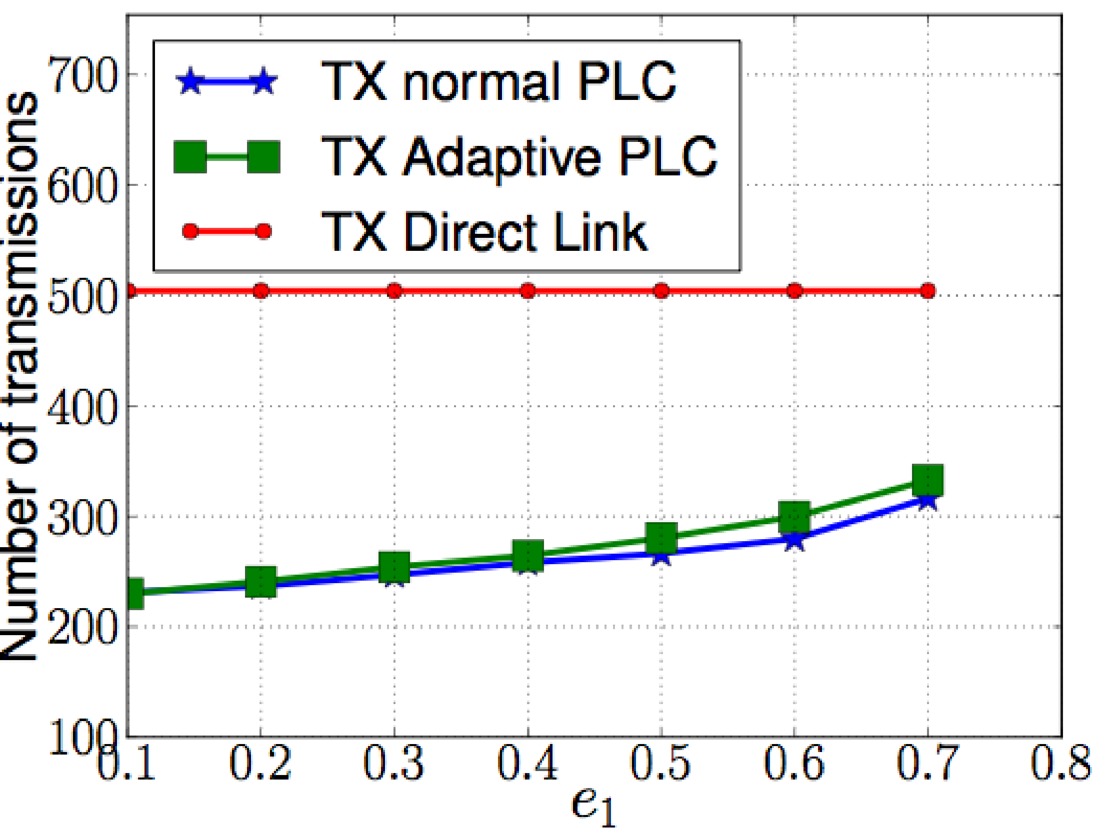
The effect the updating value

RLNC in real meshed

PlayN Cool



$e_2 = 0.3$, $e_3 = 0.8$, $g = 100$



$e_2 = 0.3$, $e_3 = 0.8$, $g = 100$

RLNC in real meshed

PlayN Cool

Avg. # of lin. dep. packets

Rx from H and S

Rx from H

4.5% overhead

DOF in destination

Avg. # of lin. dep. packets

Rx from H and S

Rx from H

1% overhead

DOF in destination

Peyman Pahlevani, Daniel E. Lucani, Frank H. P. Fitzek , “Adaptive Relay Activation in the Network Coding Protocols” *European Wireless 2015*

A Practical Guide to RLNC Libraries

S60 Implementation RLNC (2007)

Coding and decoding throughput

GF(2^8) encoding
GF(2^8) decoding
GF(2^{16}) encoding
GF(2^{16}) decoding

Throughput [kB/s]: 0, 20, 40, 60, 80, 100, 120, 140

Packets coded together: 10, 20, 50, 100, 200, 400

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

NOKIA

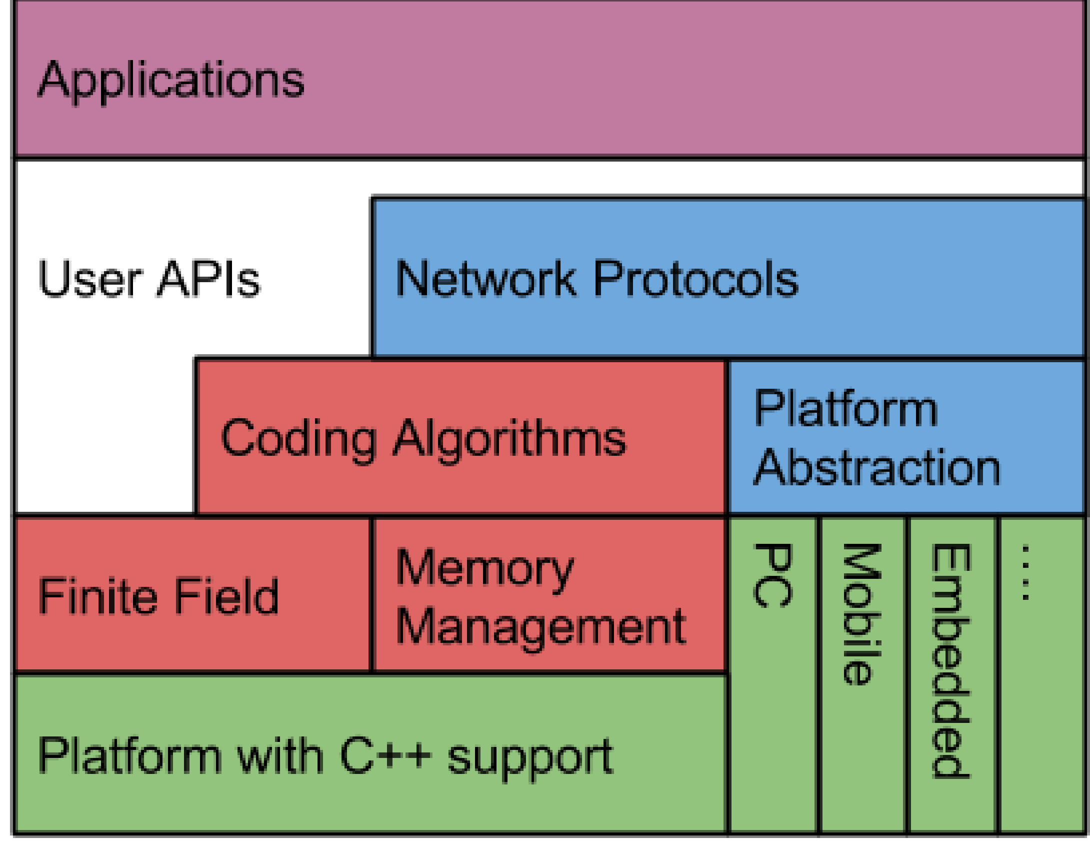
Key Technologies to Speed Up

- 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., 2^{32}-5

Software Library

Kodo: Software Implementation

- Software library for Network Coding
- Software library for Network Protocols
- Fully-tested Software
- Build System for several platforms
- In use by Customers



KODO: Shortens TTM

Implemented Features
- Recoding
- Systematic coding
- On-the-fly coding
- Partial decoding
- Real-time adjustable density
- Symbol pruning
- File encoder
- Zero copy API
- Object pooling
- Hardware optimized
- Variable symbol length

- Platforms

- Continuous Integration
  - build on every commit
  - buildbot.steinwurf.dk

Commercial Library Benchmarking (2013)

- Jerasure 1.2 by James Plank
- Jerasure 2.0 by James Plank
- OpenFEC by INRIA
- ISA-L by INTEL
- KODO by Steinwurf

- The intention is to make a fair comparison among them and start collaborative research on this topic!

Feature List (2013)

Library Capabilities	Kodo	Jerasure 1.2	Jerasure 2.0	ISA-L	Open FEC
Reed-Solomon Codes Supported	X	X	X	X	X
Network Coding Supported	X				
Updated with Novel Code Structures	X				(X)
Continuous Testing and Support	X				
Continuous Optimization of Algorithms	X				
Automatic Adaptation to CPU Features	X				
OS Support	iOS			FreeBSD	
Compiler Support	GCC, Clang, MS VS	?	GCC	GCC	?
Date of Last Release	1/2014	8/2008 12/2011[x]	1/2014	11/2013	4/2012
Hardware Acceleration on Intel Chipsets	SSSE3, CLMUL, AVX2		SSSE3	SSSE3, CLMUL	SSE
Hardware Acceleration on ARM chipsets	NEON				
Multi-core support	X				
Simulation support	Internal, NS3				

Comparison with State of the Art (2013)

Coding Speed [MB/s] for 1 MB per data segment

F=GF(2^8) P=1MB	Kodo 17 MT (sparse=0.5)	Kodo 17 (sparse=0.5)	ISA-L	Jerasure 2.0	OpenFEC
G=8 (12)	3096/2980	3096/2980	2255/2635	1250/1365	353/292
G=9 (13)	2542/2559	2752/2898	1961/2252	1096/1185	305/264
G=10 (15)	2136/2227	2025/2126	1724/1796	997/1072	285/245
G=16 (24)	1807/1496	1264/1239	1075/1180	628/644	179/160
G=30 (45)	950/647	672/513	266/271	349/361	96/90
G=60 (90)	594/329	359/256	123/122	184/184	48/46
G=100 (150)	383/209	226/159	74/73	111/111	29/28
G=150 (225)	266/141	153/107	47/46	74/74	19/19

Industry trend

RLNC (rich feature set) RS

RS use G times more memory than RLNC for data segment recovery

Measured on Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz

Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding

Benchmark
$GF(2)$
$GF(2^8)$
$GF(2^{16})$
$GF(2^{32}-5)$

Throughput [MB/s]
30
25
20
15
10
5
0
16
32
64
128
Generation Size

Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding

Benchmark chart — Throughput [MB/s] (0–30) vs. Generation Size (16, 32, 64, 128); Benchmark: $GF(2)$, $GF(2^8)$, $GF(2^{16})$, $GF(2^{32}-5)$

Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding

Throughput [MB/s]

30
25
20
15
10
5
0

Benchmark
$GF(2)$
$GF(2^8)$
$GF(2^{16})$
$GF(2^{32}-5)$

1/2

1/2

?? $An^3 + Bn^2 + Cn$??

16
32
64
128

2

Generation Size

Raspberry Pi (2013)

Hardware: 700 MHz CPU – KODO: full coding

Benchmark: $GF(2)$, $GF(2^8)$, $GF(2^{16})$, $GF(2^{32}-5)$

Throughput [MB/s]: 0, 5, 10, 15, 20, 25, 30

Generation Size: 16, 32, 64, 128

1/2

1/2

2

?? $An^3 + Bn^2 + Cn$??

$$Throughput \propto \frac{g}{t}$$

but if

$$t \propto g^2$$

then

$$Throughput \propto \frac{g}{g^2} = \frac{1}{g}$$

If we double g, we half the throughput. This shows that the component Bn^2 is more significative for small n.

Android (2013)

Hardware: 1.4 GHz CPU – KODO: full coding (using one core)

Throughput [MB/s]

90
80
70
60
50
40
30
20
10
0

16 32 64 128

Generation Size

Benchmark

$GF(2)$

$GF(2^8)$

$GF(2^{16})$

$GF(2^{32}-5)$

12:45

Thu. 3 May

PC (2013)

Hardware: 3.4 GHz CPU – KODO: full coding (using single processor)

Benchmark
$GF(2)$
$GF(2^8)$
$GF(2^{16})$
$GF(2^{32}-5)$

Throughput [MB/s]
2500
2000
1500
1000
500
0
16
32
64
128
Generation Size

KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"
        print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))

FulcrumEncoderFactoryBinary
FulcrumEncoderFactoryBinary16
FulcrumEncoderFactoryBinary4
FulcrumEncoderFactoryBinary8
FullVectorEncoderFactoryBinary
FullVectorEncoderFactoryBinary16
FullVectorEncoderFactoryBinary4
FullVectorEncoderFactoryBinary8
NoCodeEncoderFactory
OnTheFlyEncoderFactoryBinary
OnTheFlyEncoderFactoryBinary16
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8
PerpetualEncoderFactoryBinary
PerpetualEncoderFactoryBinary16
PerpetualEncoderFactoryBinary4
PerpetualEncoderFactoryBinary8
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8
SparseFullVectorEncoderFactoryBinary
SparseFullVectorEncoderFactoryBinary16
SparseFullVectorEncoderFactoryBinary4
SparseFullVectorEncoderFactoryBinary8
```

KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"
        print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))

FulcrumEncoderFactoryBinary
FulcrumEncoderFactoryBinary16
FulcrumEncoderFactoryBinary4
FulcrumEncoderFactoryBinary8
FullVectorEncoderFactoryBinary
FullVectorEncoderFactoryBinary16
FullVectorEncoderFactoryBinary4
FullVectorEncoderFactoryBinary8
NoCodeEncoderFactory
OnTheFlyEncoderFactoryBinary
OnTheFlyEncoderFactoryBinary16
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8
PerpetualEncoderFactoryBinary
PerpetualEncoderFactoryBinary16
PerpetualEncoderFactoryBinary4
PerpetualEncoderFactoryBinary8
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8
SparseFullVectorEncoderFactoryBinary
SparseFullVectorEncoderFactoryBinary16
SparseFullVectorEncoderFactoryBinary4
SparseFullVectorEncoderFactoryBinary8
```

KODO Modes

```
In [3]: # print all members containing "Factory" and "Encoder"
        print("\n".join([item for item in dir(kodo) if all([keyword in item for keyword in ["Factory", "Encoder"]])]))

FulcrumEncoderFactoryBinary
FulcrumEncoderFactoryBinary16
FulcrumEncoderFactoryBinary4
FulcrumEncoderFactoryBinary8
FullVectorEncoderFactoryBinary
FullVectorEncoderFactoryBinary16
FullVectorEncoderFactoryBinary4
FullVectorEncoderFactoryBinary8
NoCodeEncoderFactory
OnTheFlyEncoderFactoryBinary
OnTheFlyEncoderFactoryBinary16
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8
PerpetualEncoderFactoryBinary
PerpetualEncoderFactoryBinary16
PerpetualEncoderFactoryBinary4
PerpetualEncoderFactoryBinary8
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8
SparseFullVectorEncoderFactoryBinary
SparseFullVectorEncoderFactoryBinary16
SparseFullVectorEncoderFactoryBinary4
SparseFullVectorEncoderFactoryBinary8
```

Network Coding GF(2)

(a)

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ & & & \end{pmatrix}$$

(b)

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \end{pmatrix}$$

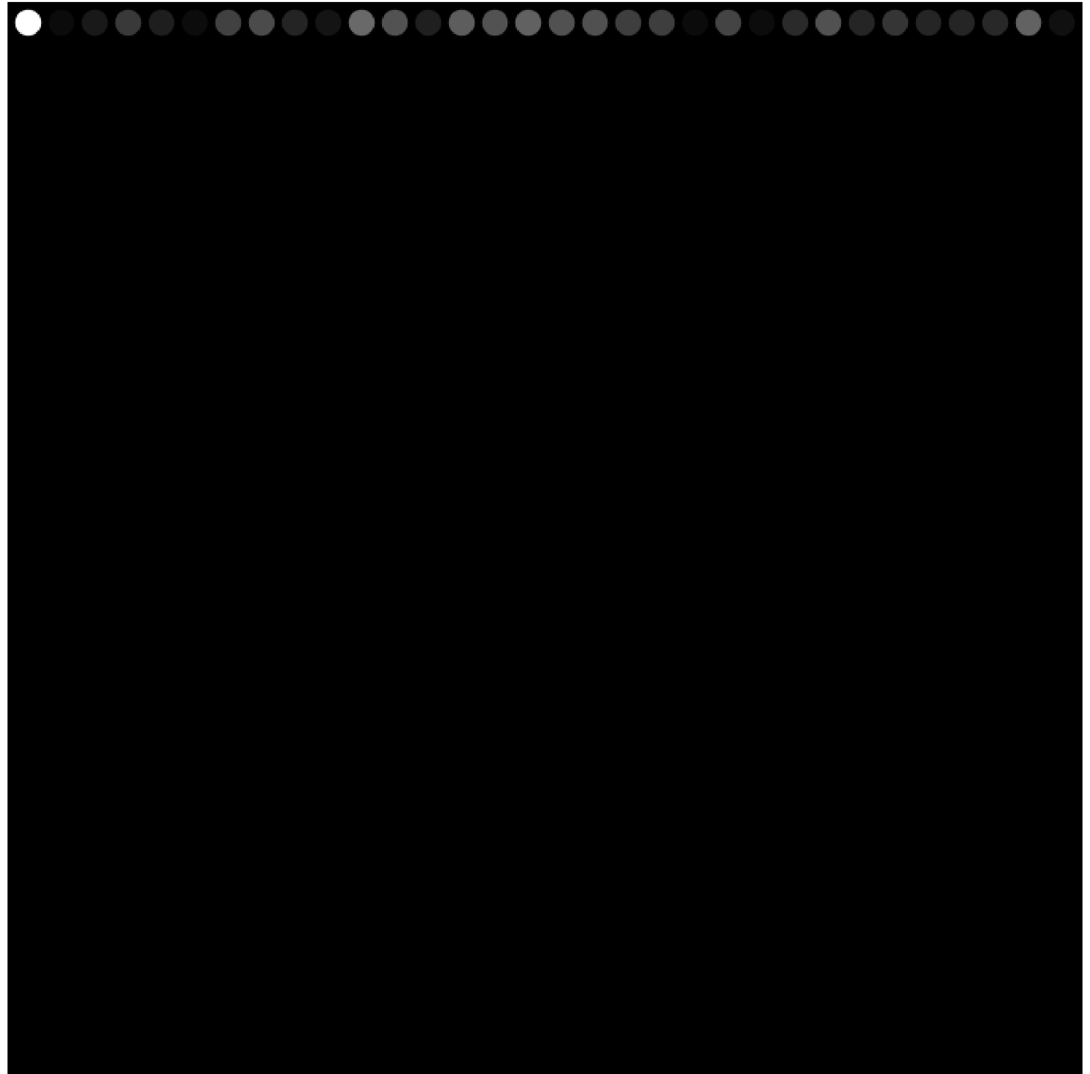
(c)

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \end{pmatrix}$$

FullRLNC

Coding matrix is loaded with fully random elements of field size F

Probability of zero as field element is 1/F



Systematic RLNC

Picture Viewer
Pict...
Options Back

(d)

Picture Viewer
Pict...
Options Back

(e)

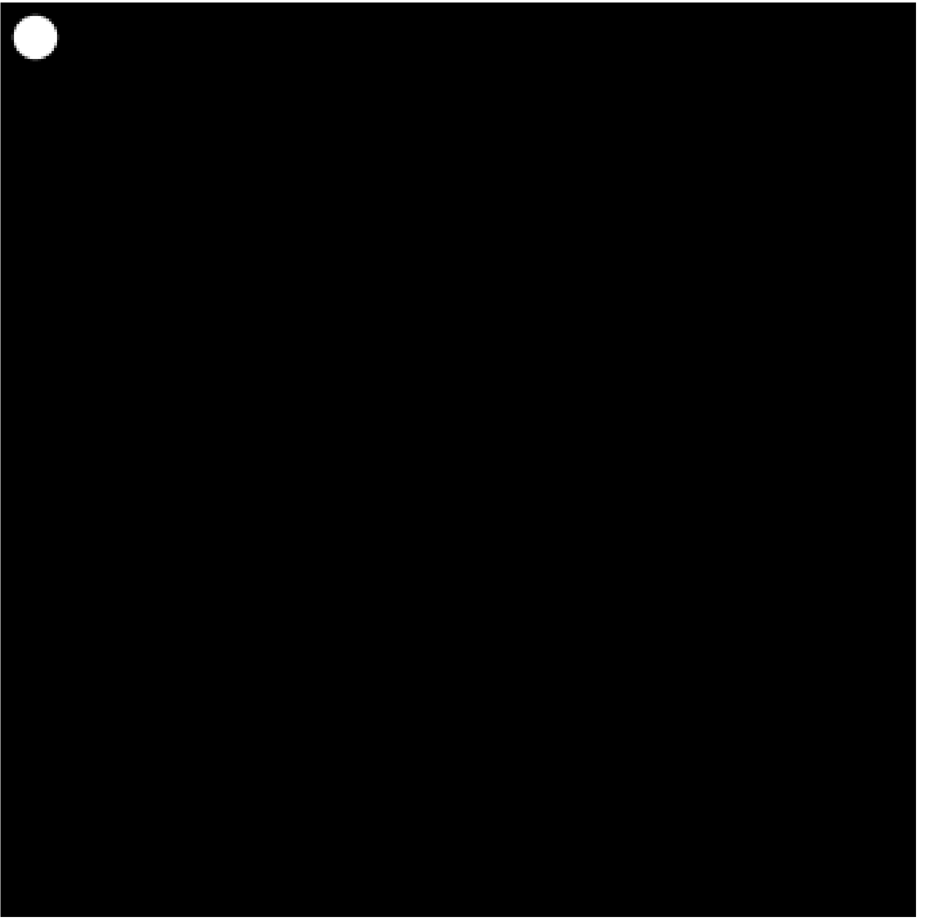
Picture Viewer
Pict...
Options Back

(f)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} \\ 0 & 0 & 1 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} \end{pmatrix}$$

Systematic RLNC

Starting with uncoded packets and fill wholes with fully encoded packets



KODO: RLNC Modes

KODO: RLNC Modes

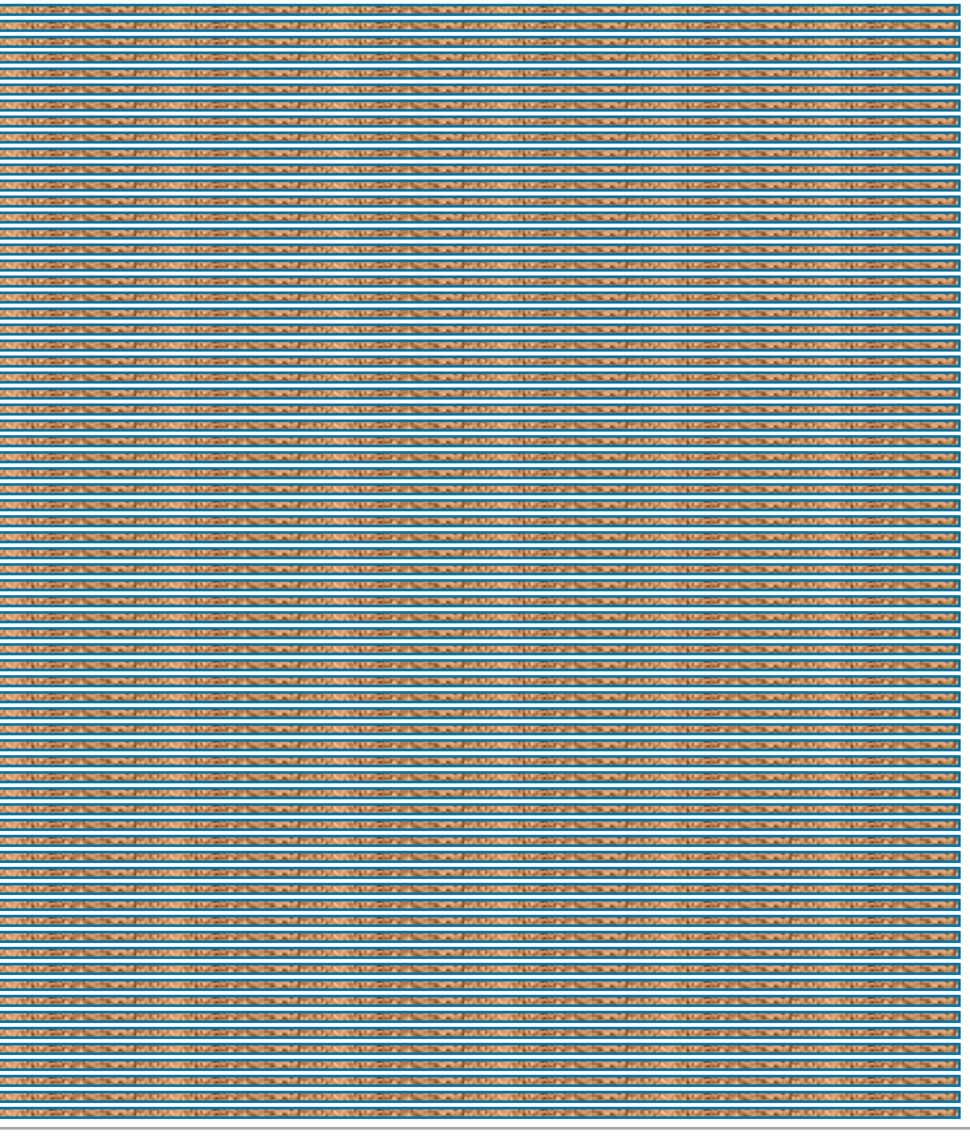
Input for Coding

KODO: RLNC Modes

Basic Idea

KODO: RLNC Modes

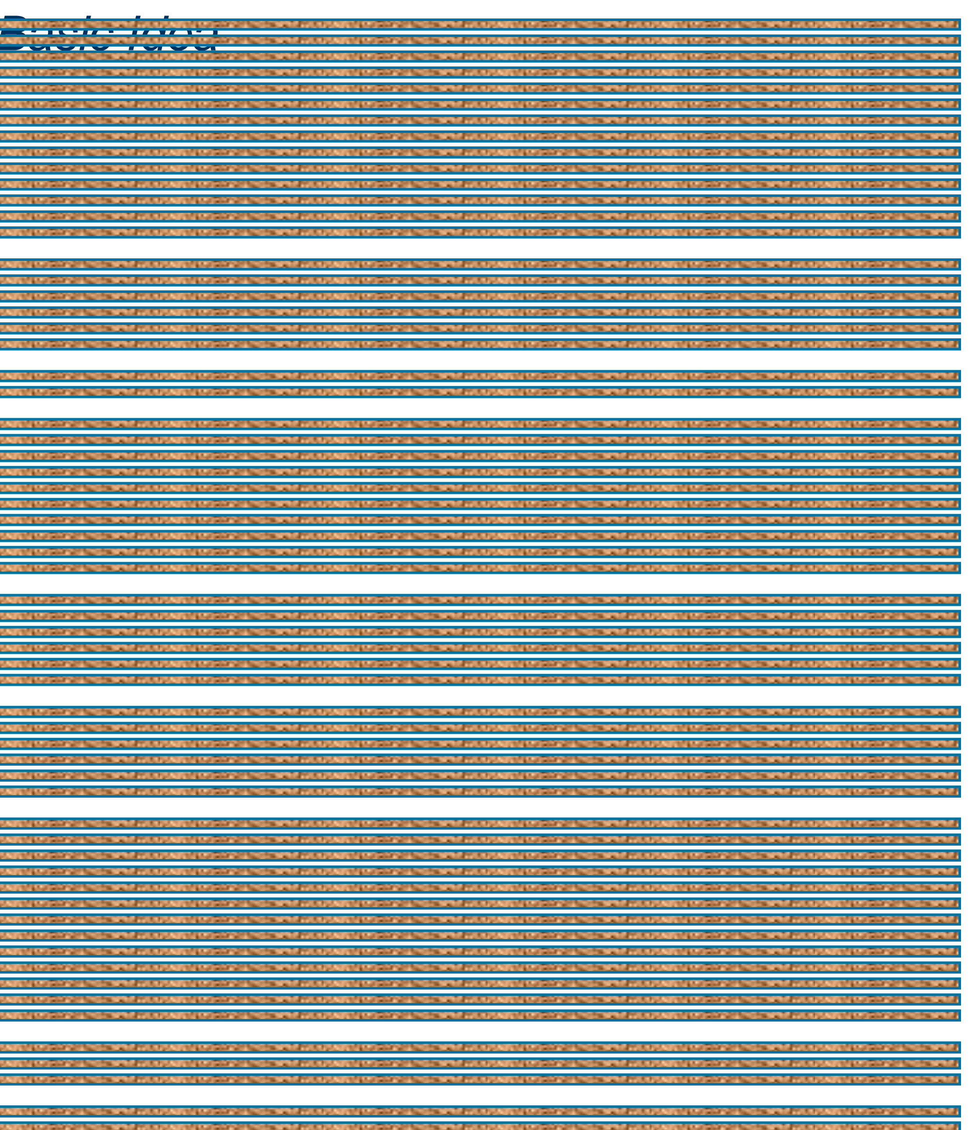
Basic Idea



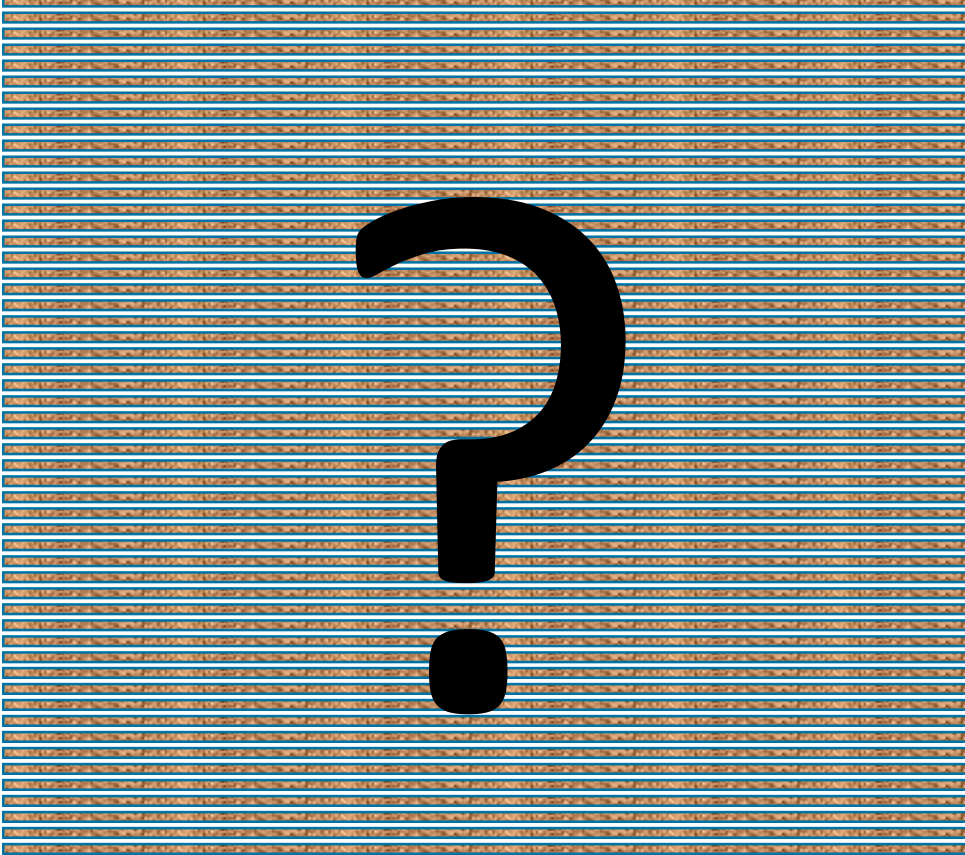
KODO: RLNC Modes

Basic Idea

Lossy Medium



KODO - Decoder



KODO: RLNC Modes

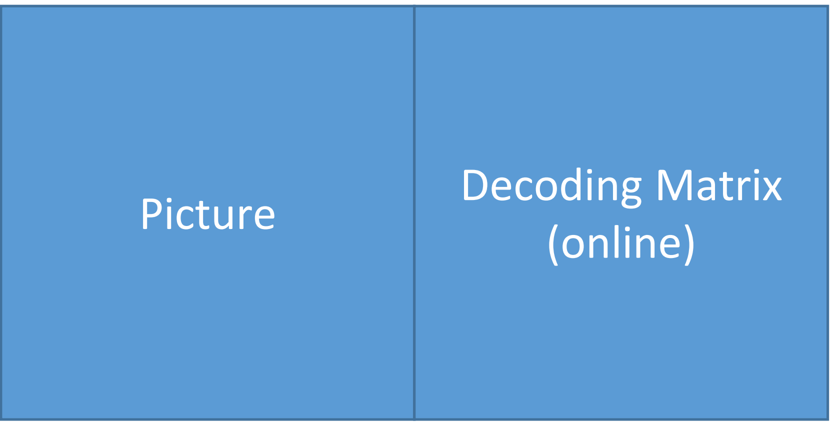
Basic Idea

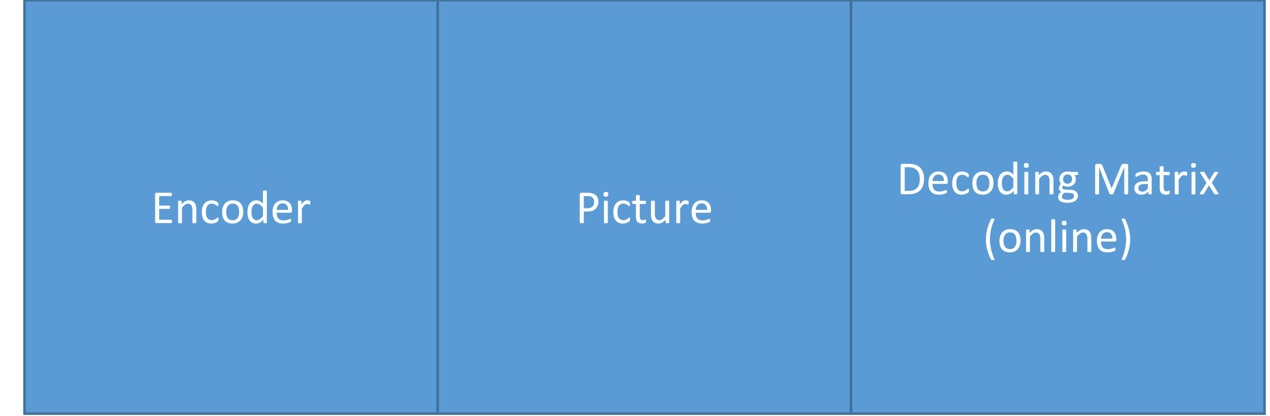
Modes

- Systematic ON/OFF
- On the fly / Online
- Sliding Window

Parameters

- Field 2, 2^8, 2^{16}
- Input (Salta vs. Lena vs. Pikachu)





Full RLNC

```
FullVectorEncoderFactoryBinary
FullVectorEncoderFactoryBinary16
FullVectorEncoderFactoryBinary4
FullVectorEncoderFactoryBinary8
```

Salta – Non Systematic Full RLNC

Finite Field : 2 - Binary

Salta – Non Systematic Full RLNC

Finite Field : Binary8

Systematic RLNC

Systematic RLNC

- In general, Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations

$$
\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}
$$

Systematic RLNC

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

Nothing to do here!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

Nothing to do here!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to "clean" up the matrix

Substract first row multiplied with $\alpha_{4,1}$ from fourth line!
Substract first row multiplied with $\alpha_{5,1}$ from fifth line!
Substract first row multiplied with $\alpha_{6,1}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ \alpha_{5,1} & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

Substract second row multiplied with $\alpha_{4,2}$ from fourth line!
Substract second row multiplied with $\alpha_{5,2}$ from fifth line!
Substract second row multiplied with $\alpha_{6,2}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & \alpha_{4,2} & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & \alpha_{6,2} & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to "clean" up the matrix

Substract third row multiplied with $\alpha_{4,3}$ from fourth line!
Substract third row multiplied with $\alpha_{5,3}$ from fifth line!
Substract third row multiplied with $\alpha_{6,3}$ from sixth line!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & 0 & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & 0 & \alpha_{6,3} & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Systematic RLNC

- Gaussian elimination n x n matrix requires $An^3 + Bn^2 + Cn$ operations
- Use D uncoded information to “clean” up the matrix

M-D

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & 0 & 0 & \alpha_{5,4} & \alpha_{5,5} & \alpha_{5,6} \\ 0 & 0 & 0 & \alpha_{6,4} & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

$A(M-D)^3 + B(M-D)^2 + C(M-D)$

M

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

$A(M)^3 + B(M)^2 + C(M)$

Lesson Learned

- Binary fields yield high coding rates (simple XOR calculations and 50% of the coefficients are zero)
- Systematic approach also yields high coding rates
- It seems the number of zeros in the coding matrix plays a role here. Why?
- Zeros allows for reordering of the coded packets rather than brute force calculating. Among other reasons.
- Remember some slides before:

0110	*1000 ← Simple MOVE*	*1000*
1000	*0110*	*0101 (0110 XOR 0011)*
1101	*0011 ← 0101 (XOR 0110) ← 1101 (XOR 1000)*	*0011*

Sparse RLNC

```
SparseFullVectorEncoderFactoryBinary
SparseFullVectorEncoderFactoryBinary16
SparseFullVectorEncoderFactoryBinary4
SparseFullVectorEncoderFactoryBinary8
```

Sparse Network Codes

- Let‘s add zeros randomly into our coding matrix.

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & 0 & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & \alpha_{4,3} & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & 0 \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

- Let‘s add zeros randomly into our coding matrix. More!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & 0 \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,4} & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & \alpha_{4,5} & \alpha_{4,6} \\ 0 & \alpha_{5,2} & \alpha_{5,3} & \alpha_{5,4} & \alpha_{5,5} & 0 \\ \alpha_{6,1} & \alpha_{6,2} & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

- Let's add zeros randomly into our coding matrix. More! More!

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & 0 & \alpha_{1,6} \\ \alpha_{2,1} & 0 & \alpha_{2,3} & 0 & \alpha_{2,5} & 0 \\ 0 & 0 & \alpha_{3,3} & 0 & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & 0 & \alpha_{4,6} \\ 0 & \alpha_{5,2} & 0 & \alpha_{5,4} & 0 & 0 \\ \alpha_{6,1} & 0 & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

≈ 300 MB/s
≈ 33 MB/s
≈ 20 MB/s
≈ 0.4 MB/s

Generation Size = 32 Packets
Generation Size = 256 Packets
Generation Size = 2048 Packets

Decoder Processing Speed [MB/s]

10^{3}, 10^{2}, 10^{1}, 10^{0}, 10^{-1}

0 5 10 15 20 25 30 35

More dense

Mean Number of Non-Zero Coefficients (Density)

One or two orders of magnitude in the coding speed by

Dualism Theory and Implementation!

Sparse Network Codes

- Let‘s add zeros randomly into our coding matrix. More! More!
- The more zeros we add the fast the decoding process.
- BUT, what about linear dependency?

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} 0 & \alpha_{1,2} & 0 & \alpha_{1,4} & 0 & \alpha_{1,6} \\ \alpha_{2,1} & 0 & \alpha_{2,3} & 0 & \alpha_{2,5} & 0 \\ 0 & 0 & \alpha_{3,3} & 0 & 0 & \alpha_{3,6} \\ \alpha_{4,1} & 0 & 0 & \alpha_{4,4} & 0 & \alpha_{4,6} \\ 0 & \alpha_{5,2} & 0 & \alpha_{5,4} & 0 & 0 \\ \alpha_{6,1} & 0 & \alpha_{6,3} & 0 & \alpha_{6,5} & \alpha_{6,6} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{pmatrix}$$

Sparse Network Codes

Can we operate in the “speed up” region while maintaining good performance?

Yes, but the key is not to use a fixed density

(Tunable sparse network coding, 2012)

GF(2)

Simulation with 100,000 generations

Number of Coded Packets Received

Mean Number of Non-Zero Coefficients (Density)

Maximum
Minimum
Mean

Tunable Sparse RLNC

Tunable Sparse Network Codes

Scheme with feedback: targets specific performance degradation

NonzeroTsncDofFeedback, budget: 8, n-sparse: 12.67
NonzeroTsncDofFeedback, budget: 32, n-sparse: 18.83
NonzeroTsncDofFeedback, budget: 128, n-sparse: 16.58

Density

Decoder rank

Tunable Sparse Network Codes

Coder_type: decoder, Field: Binary8

Goodput [MB/s]

600
500
400
300
200
100
0

0 50 100 150 200 250

Dependent packets

2
4
8
16
32
64
128

- FullRLNC, n-sparse: 0.0
- FullRLNC, n-sparse: 0.0
- NonzeroTsncDofFeedback, n-sparse: 0.0
- SparseNonzeroFullRLNC, n-sparse: 2.0
- SparseNonzeroFullRLNC, n-sparse: 4.0
- SparseNonzeroFullRLNC, n-sparse: 8.0
- SparseNonzeroFullRLNC, n-sparse: 16.0

Tunable Sparse
~ 410MB/s

Tunable Sparse
~170MB/s

RLNC
~50MB/s

Recoding for Sparse RLNC

- So what about recoding of sparse RLNC packets?
- In general yes it is possible.
- But by agnostic recoding the characteristics in terms of sparsity of the packets will change.

Perpetual RLNC

```
PerpetualEncoderFactoryBinary
PerpetualEncoderFactoryBinary16
PerpetualEncoderFactoryBinary4
PerpetualEncoderFactoryBinary8
```

Perpetual RLNC

- So increasing the number of the zeros in the coding matrix with the right amount can help in terms of complexity.
- The length of the encoding vector G * $\log_2(F)$ can be decreased if the encoding vector is sparse (larger number of zeros).
  - Variable Length Coding (VLC)
  - Huffman Coding
- The idea of perpetual RLNC is to use the same number of non-zero coefficient but in a more planned manner instead of randomly spreading over the coding matrix.
  - Phase I: number of zeros
  - Phase II: number of non-zero elements
  - Phase III: number of zeros
- Encoding vector will be even smaller

Perpetual RLNC

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

Perpetual RLNC

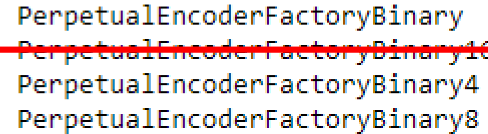
$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \end{pmatrix} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,4} & \alpha_{1,5} & \alpha_{1,6} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,4} & \alpha_{2,5} & \alpha_{2,6} \\ \alpha_{3,1} & \alpha_{3,2} &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

Perpetual RLNC

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

Perpetual RLNC

Encoder	Content	Decoder



On the Fly RLNC

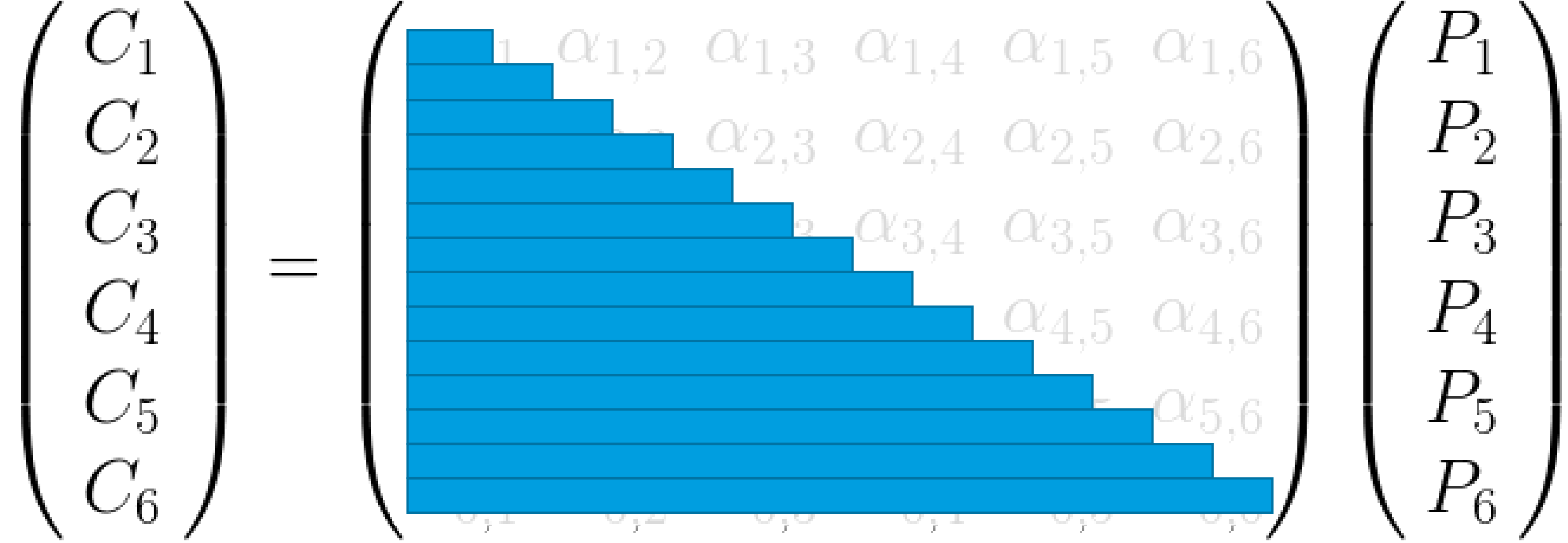
```
OnTheFlyEncoderFactoryBinary
OnTheFlyEncoderFactoryBinary16
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8
```

On the Fly RLNC

- Instead of a fixed corridor, On the Fly RLNC is encoding incoming packets on the fly.
- This could lead to instantaneous decodability at the receiver if no error occurs

On the Fly RLNC

agnostic approach



On the Fly RLNC

agnostic approach - error free

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_1+P_2+P_3+P_4+P_5$

$P_1+P_2+P_3+P_4+P_5+P_6$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7+P_8$

P_1

P_2

P_3

P_4

P_5

P_6

P_7

P_8

On the Fly RLNC

agnostic approach - error prone

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_1+P_2+P_3+P_4+P_5$

$P_1+P_2+P_3+P_4+P_5+P_6$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7+P_8$

P_1

P_2

1

2

3

4

On the Fly RLNC

agnostic approach - error prone

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_1+P_2+P_3+P_4+P_5$

$P_1+P_2+P_3+P_4+P_5+P_6$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7+P_8$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7+P_8$

$P_1+P_2+P_3+P_4+P_5+P_6+P_7+P_8$

1

2

3

4

5

6

P_1

P_2

P_3 P_4 P_5 P_6 P_7 P_8

On the Fly RLNC

error aware approach

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

On the Fly RLNC

Encoder	Content	Decoder

```
OnTheFlyEncoderFactoryBinary
~~OnTheFlyEncoderFactoryBinary16~~
OnTheFlyEncoderFactoryBinary4
OnTheFlyEncoderFactoryBinary8
```

Sliding Window RLNC

```
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8
```

Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback

Sliding Window

with feedback

ACK

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

Sliding Window

with feedback

NACK

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

Sliding Window

with feedback

ACK

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

Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback
- Opening the window on ACK reception
- Closing the window to reduce complexity

Sliding Window

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

Sliding Window

- So far we estimated the channel errors
- Would could we do if we would know about errors that happen?
- First of all we need a feedback
- Opening the window on ACK reception
- Closing the window to reduce complexity
- Width of the window depends on propagation delay and error characteristics

Sliding Window

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_2+P_3+P_4+P_5$

$P_3+P_4+P_5+P_6$

$P_4+P_5+P_6+P_7$

$P_5+P_6+P_7+P_8$

width

P_1

P_2

P_3

P_4

P_5

P_6

P_7

P_8

Sliding Window

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_2+P_3+P_4+P_5$

$P_2+P_3+P_4+P_5$

$P_3+P_4+P_5+P_6$

$P_4+P_5+P_6+P_7$

width

1

2

3

P_1

delay

P_2 P_3 P_4 P_5

P_6

P_7

Sliding Window

encoder

decoder

P_1

P_1+P_2

$P_1+P_2+P_3$

$P_1+P_2+P_3+P_4$

$P_2+P_3+P_4+P_5$

$P_2+P_3+P_4+P_5$

$P_3+P_4+P_5+P_6$

$P_4+P_5+P_6+P_7$

width

P_1

P_2

P_3

P_4

P_5

P_6

P_7

P_8

Sliding Window

encoder

decoder

P_1

P_2

P_2+P_3

P_2+P_3

P_4

P_4+P_5

$P_4+P_5+P_6$

P_7

width

P_1

P_2 P_3

P_4

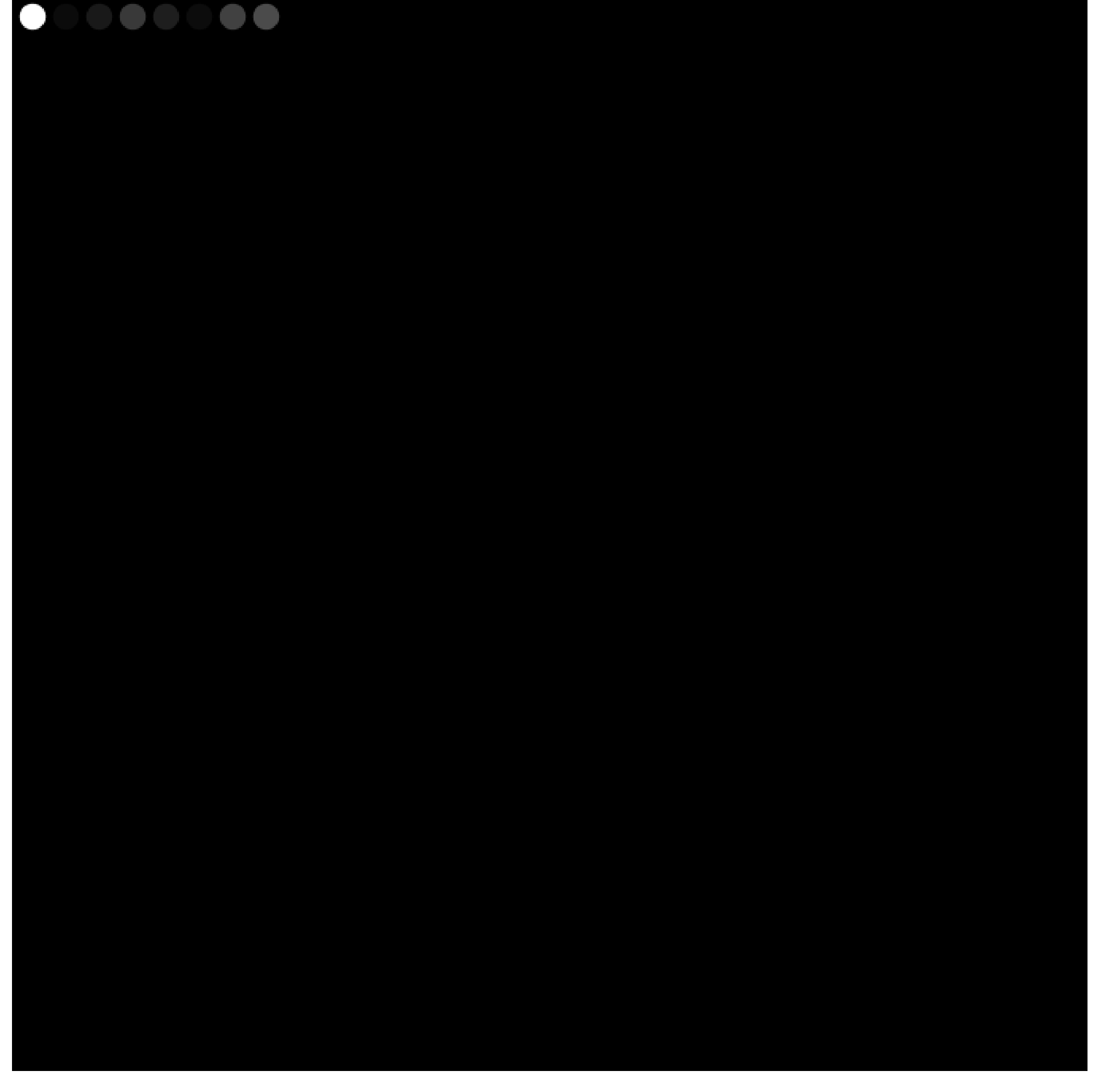
P_5

P_6

P_7

Sliding Window

- Terminology
  - On the fly encoding/decoding: Newly incoming packets are added to the block.
  - Sliding window: Newly incoming packets are added to the block and acknowledged packets are removed.
  - Sliding window can work without blocks



Sliding Window

Encoder	Content	Decoder

```
SlidingWindowEncoderFactoryBinary
SlidingWindowEncoderFactoryBinary16
SlidingWindowEncoderFactoryBinary4
SlidingWindowEncoderFactoryBinary8
```

Sliding Window

Steinwurf's Example ARQ vs. Coding

Fulcrum RLNC

```
FulcrumEncoderFactoryBinary
FulcrumEncoderFactoryBinary16
FulcrumEncoderFactoryBinary4
FulcrumEncoderFactoryBinary8
```

Problem: Heterogenity in Display Size

ng Galaxy Tab 3 10.1 P5200
243.1 x 176.1

ng Galaxy Tab 3
8.0
9.8 x 123.8

Galaxy Tab 3
1 P5200
x 111.1

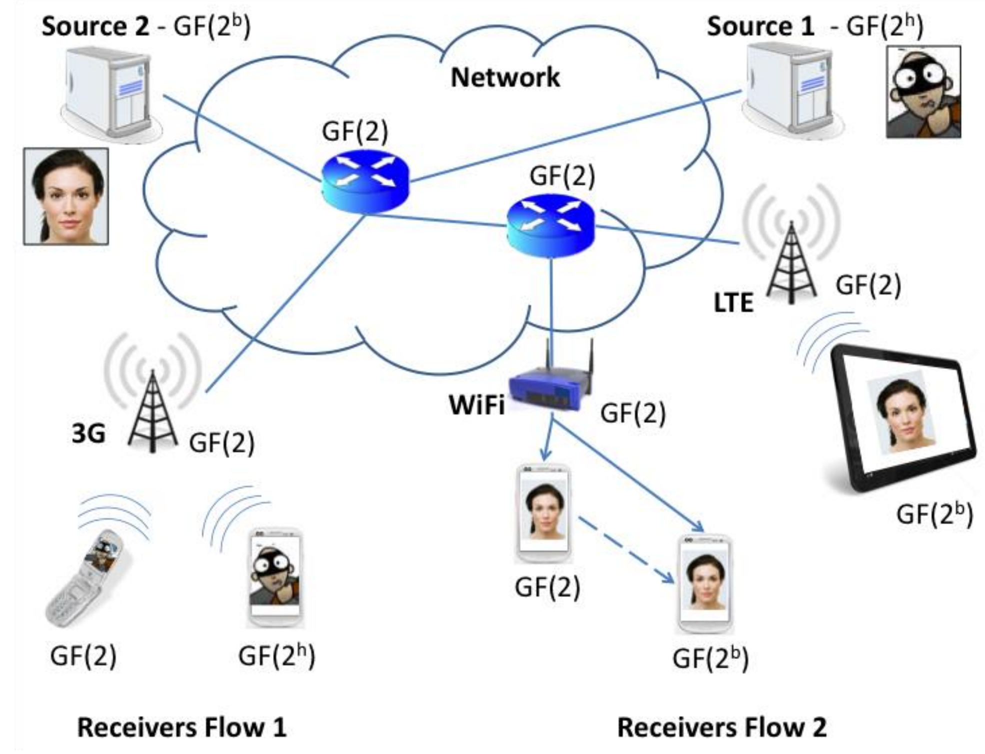
ung
Note II
00
80.5

ng
laxy
ve
71.3

Samsung
Galaxy Ace
3
121.2 x
62.7

General Ideas

- Fluid allocation of complexity
- End devices agree on desired performance:
  - Independent from network
  - Chosen according to application requirements
- Network devices need only support a simple subset of functions
- Reduces overhead
  - Roughly 1 bit per coding coefficient
- Key: code concatenation with different field sizes



Benefits

- Simple is green, compatible, deployable
- Supports heterogeneous receivers
- Adaptive performance
- Practical recoding
- Spreads complexity to stronger devices
- Security: simple support
- New designs can be supported: backwards compatible

General Structure

Sender application

Outer encoder

Inner encoder

Source

(optional)

Outer recoder

Inner recoder

Network

Combined decoding

Outer decoding

Inner decoding

Discard expansion

Inner decoder

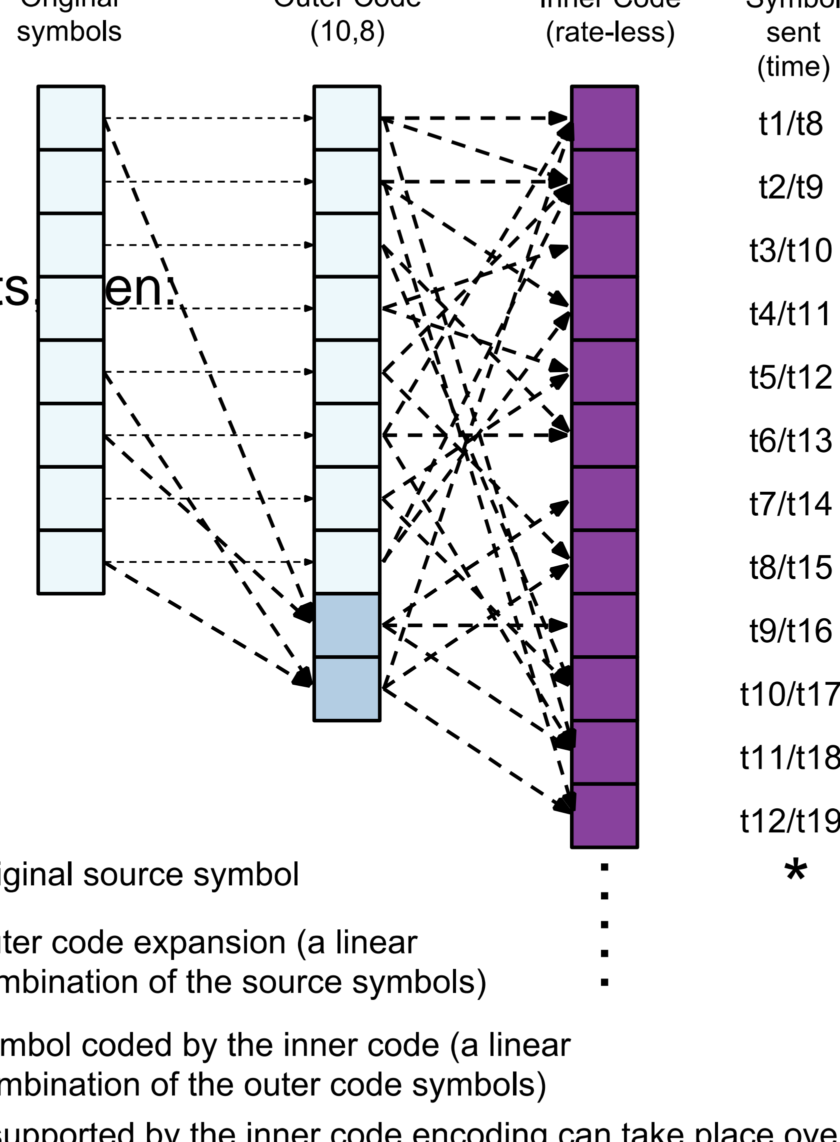
Receiver application

Sink

- Inner code: RLNC, Sparse RLNC, Perpetual, … GF(2)
- Outer code: (systematic) RLNC, Reed-Solomon, … GF(2^h)

Encoder

- Since inner code is created in GF(2) with pre-coded packets, then:
- n+r coefficients per inner coded packet
- This means: 1 + r/n bits per coefficient per original packet



Encoder

Outer code coefficient matrix

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
5	22	18	4	73
54	92	32	42	2

Mapping outer code to inner

1 * Packet 1
+
0 * Packet 2
+
0 * Packet 3
+
0 * Packet 4
+
0 * Packet 5

Inner code initialization

Packet 1

1
0
0
0
0
0
0

1 bit

n + r bits

Encoder

Outer code coefficient matrix

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
5	22	18	4	73
54	92	32	42	2

Mapping outer code to inner

0 * Packet 1
+
0 * Packet 2
+
0 * Packet 3
+
0 * Packet 4
+
1 * Packet 5

Inner code initialization

Packet 5

0
0
0
0
1
0
0

Encoder

Outer code coefficient matrix

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
5	22	18	4	73
54	92	32	42	2

Mapping outer code to inner

5 * Packet 1
+
22 * Packet 2
+
18 * Packet 3
+
4 * Packet 4
+
73 * Packet 5

Inner code initialization

Packet 6

0
0
0
0
0
1
0

Encoder

Outer code coefficient matrix

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
5	22	18	4	73
54	92	32	42	2

Mapping outer code to inner

54 * Packet 1
+
92 * Packet 2
+
32 * Packet 3
+
42 * Packet 4
+
2 * Packet 5

Inner code initialization

Packet 7

0
0
0
0
0
0
1

Received Packets before Decoding

Probability of Decoding

Number of Coded Packets Received

	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10

Legend:
- Standard GF(2)
- r = 1 extra dimension
- r = 2 extra dimensions
- r = 3 extra dimensions
- r = 4 extra dimensions
- r = 5 extra dimensions
- r = 6 extra dimensions
- r = 7 extra dimensions

r = 0, 1,..., 7

Higher r: Close to high-field performance

Coded Packets Sent
Mean: $n + O(2^{-r})$
Variance: $O(2^{-r})$

Received Packets before Decoding

Code		Decoding after receiving (coded packets)			
		n	n + 1	n + 2	n + 3
Fulcrum	r = 4	93.87%	99.75%	99.99%	99.9997%
	r = 7	99.22%	99.996%	99.99998%	99.99999992%
	r = 10	99.90%	99.9999%	99.99999996%	99.99999999998%
RaptorQ*		99%	99.99%	99.9999%	

* Qualcomm. (2013, Dec.) Raptorq - the superior fec technology
Available: http://www.qualcomm.com/media/documents/raptorq-data-sheet

Performance Results: Encoder

Processing speed encoder [MB/s]

2000
1000
400
200
100
50
20
10
0.8
0.4

16 32 64 128 256 512 1024

Generation Size [Packets]

>5x

>17x

25x

RLNC Binary
RLNC Binary8
FulcrumCombined r = 1
FulcrumCombined r = 2
FulcrumCombined r = 3
FulcrumCombined r = 4

Performance Results: Decoder

Processing speed decoder [MB/s]

2000
1000
100
50
20
10
1
0.5

16 32 64 128 256 512 1024

Generation Size [Packets]

>2x
>7x
>20x

RLNC Binary
RLNC Binary8
FulcrumCombined r = 1
FulcrumCombined r = 4
FulcrumInner r = 1
FulcrumInner r = 4
FulcrumOuter r = 1
FulcrumOuter r = 4

Advantages

- Low overhead:
  - 1 + r/n ~ 1 bit per coefficient per packet → like GF(2)
  - Total transmitted packets: n + O(2-r) → like higher fields
- Processing speed (complexity) compared to GF(28):
  - Encoder 5x to 25x faster
  - Decoder 2x to >20x faster
- Supports heterogeneous receivers
- Allows a fluid allocation of complexity
- Simple security support
- Network can implement a bare minimum:
  - Just XOR packets!

Heterogeneous Users

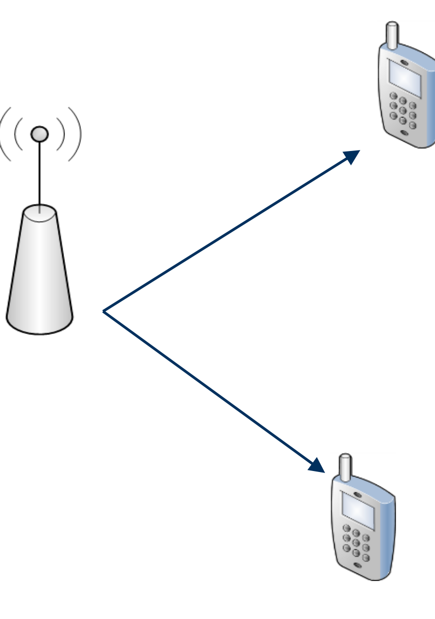
Fulcrum – Both Rx use outer decoder, r = 2
Fulcrum – Both Rx use outer decoder, r = 7
Fulcrum – One inner and one outer decoder, r = 2
Fulcrum – One inner and one outer decoder, r = 7
Both Rx use RLNC GF(2)
Both Rx use RLNC GF(2^{16})

Completion Time (CDF)

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

10 12 14 16 18 20 22 24 26 28 30

Number of Transmissions



Composite Extension Finite Fields for Low Overhead Network Coding: Telescopic Codes

Nestor Hernandez, Janus Heide, Daniel Enrique Lucani Roetter, and Frank Hanns Paul Fitzek, “On the Overhead of Telescopic Codes in Network Coded Cooperation,” I E E E V T S Vehicular Technology Conference. Proceedings, 2015.

Introduction

- Network coding is a promising paradigm
  - Store-code-forward instead of store-forward
  - Benefits in throughput, delay, robustness, energy
- Caveats:
  - Computational complexity
  - Overhead per packet, e.g., signaling of coding coefficients

Motivation

Probability of Decoding the Original Packets

Number of Received Coded Packets

GF(2)
GF(2^2)
GF(2^8)

Throughput [MB/s]

Generation Size

Benchmark
GF(2)
GF(2^8)
GF(2^{16})
GF($2^{32}-5$)

Larger fields are more efficient!

?

Smaller fields are faster and need less overhead!

Motivation

Kodo Simulations cdda39b-dirty

Simulation | Rank diagram | 3D loss diagram | Recoding strategies diagram

Number of rounds: 500

Symbols: 16

Field size:
- [x] BINARY
- [x] BINARY8
- [ ] BINARY16
- [ ] PRIM2325

Number of linear dependent packets between each rank increase

Retransmissions (y-axis): 0,00 0,05 0,10 0,15 0,20 0,25 0,30 0,35 0,40 0,45 0,50 0,55 0,60 0,65 0,70 0,75 0,80 0,85 0,90

Rank (x-axis): 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Legend: GF(2^8) GF(2)

Motivation

Kodo Simulations cdda39b-dirty

Simulation | Rank diagram | 3D loss diagram | Recoding strategies diagram

Number of rounds: 500
Symbols: 16
Field size: BINARY (checked), BINARY8 (checked), BINARY16, PRIM2325

Number of linear dependent packets between each rank increase

Retransmissions (y-axis: 0,00 – 0,90); Rank (x-axis: 0 – 15)

Legend: GF(2^8) GF(2)

GF(2) has a problem in the end!

GF(2^8) has nearly no problem in the end!

Introduction

Previous work:

- Intrinsic information conveyance in network coding by Heide/Pedersen/Fitzek/Zhang
- WO2011114191A1
- Understanding single choice of field size, generation size on processing speed [Paramanathan et al 2013]
- Techniques that exploit sparsity: tunable sparse network coding, overlapping generations, BATS codes, ...
- Fulcrum network codes: low overhead, high processing speed, simple recoding, configurable performance [Lucani et al 2014]

Motivation

Fulcrum provides support using composite fields:

GF(2) in the network and any GF(2^k) at source and destinations

Key: operations performed in GF(2) have an equivalent operation in any GF(2^k)

Example: P1 + P2 requires a bit by bit XOR in both cases

Can we expand this idea for more flexibility in code design?

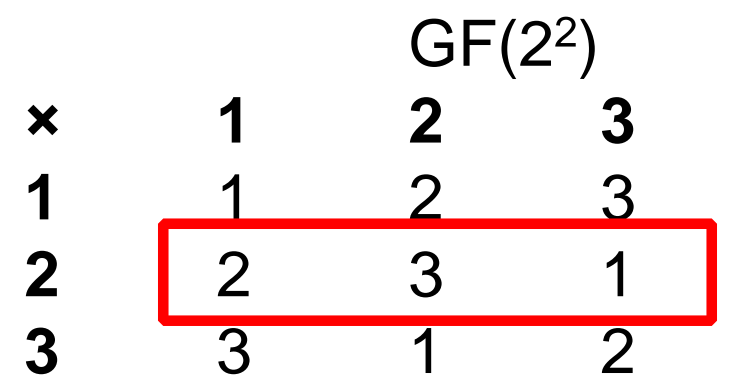
Typically, GF(2^k) is not compatible with a GF(2^p) if $n \neq p$

Different primitive polynomials → different mapping for product operations

Motivation

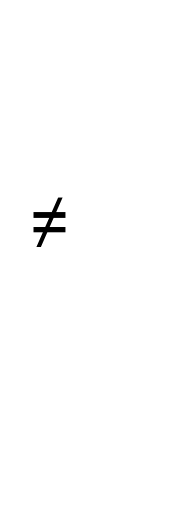
GF(2^2) polynomial: X^2 + X + 1
GF(2^4) polynomial: X^4 + X + 1

- Typically, $GF(2^k)$ is not compatible with a $GF(2^p)$ if n ≠ p
- Example: $GF(2^2)$ versus $GF(2^4)$



$GF(2^2)$

×	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2



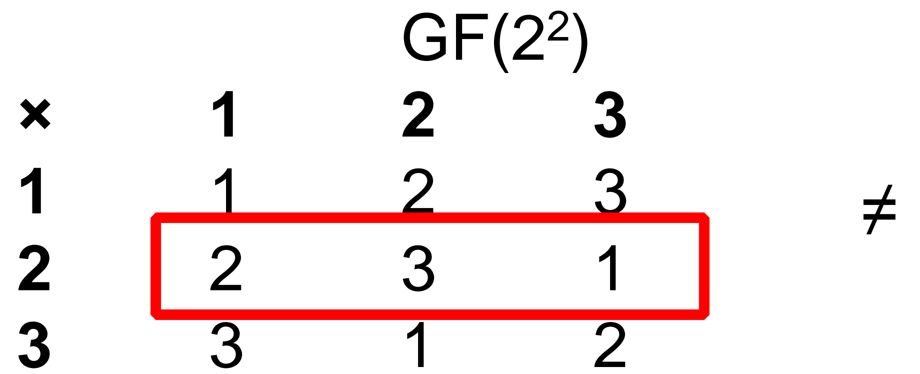
$GF(2^4)$

x	1	2	3	4	5	…	15
1	1	2	3	4	5	…	15
2	2	4	6	8	10	…	13
3	3	6	5	12	15	…	2
4	4	8	12	3	7	…	9
5	5	10	15	7	2	…	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	13	2	9	6	…	10

Motivation

GF(2^2) polynomial: X^2 + X + 1
GF(2^4) polynomial: X^4 + X + 1

- Typically, $GF(2^k)$ is not compatible with a $GF(2^p)$ if $n \neq p$
- Example: $GF(2^2)$ versus $GF(2^4)$



$GF(2^2)$

×	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

$\neq$

$GF(2^4)$

x	1	2	3	4	5	…	15
1	1	2	3	4	5	…	15
2	2	4	6	8	10	…	13
3	3	6	5	12	15	…	2
4	4	8	12	3	7	…	9
5	5	10	15	7	2	…	6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	13	2	9	6	…	10

- Actually, we will need even stronger conditions
- Idea: Multiple composite extension fields

Multiple Composite Extension Fields

Use GF(2) to construct GF(22), use GF(22) to construct GF(222)

GF(2^2)

×	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

GF(2^{2^2})

x	1	2	3	4	5	…	15
1	1	2	3	4	5	…	15
2	2	3	1	8		…	5
3	3	1	2	12		…	10
4	4	8	12	6		…	1
5	5					…	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	5	10	1		…	9

Full compliance: product by
1, 2, 3 in GF(2^{2^2}) (4 bits)
Is equivalent to the product on two values of
GF(2^2) (2 bits)

Multiple Composite Extension Fields

Use GF(2) to construct $GF(2^2)$, use $GF(2^2)$ to construct $GF(2^{2^2})$

Example: 15 x 2 = 5

$1111_b \times 0010_b$ in $GF(2^{2^2})$

But in $GF(2^2)$ we operate on pairs of bits

$11_b \times 10_b = 01_b$

Thus, on a vector

$11_b11_b \times 10_b = 01_b01_b$

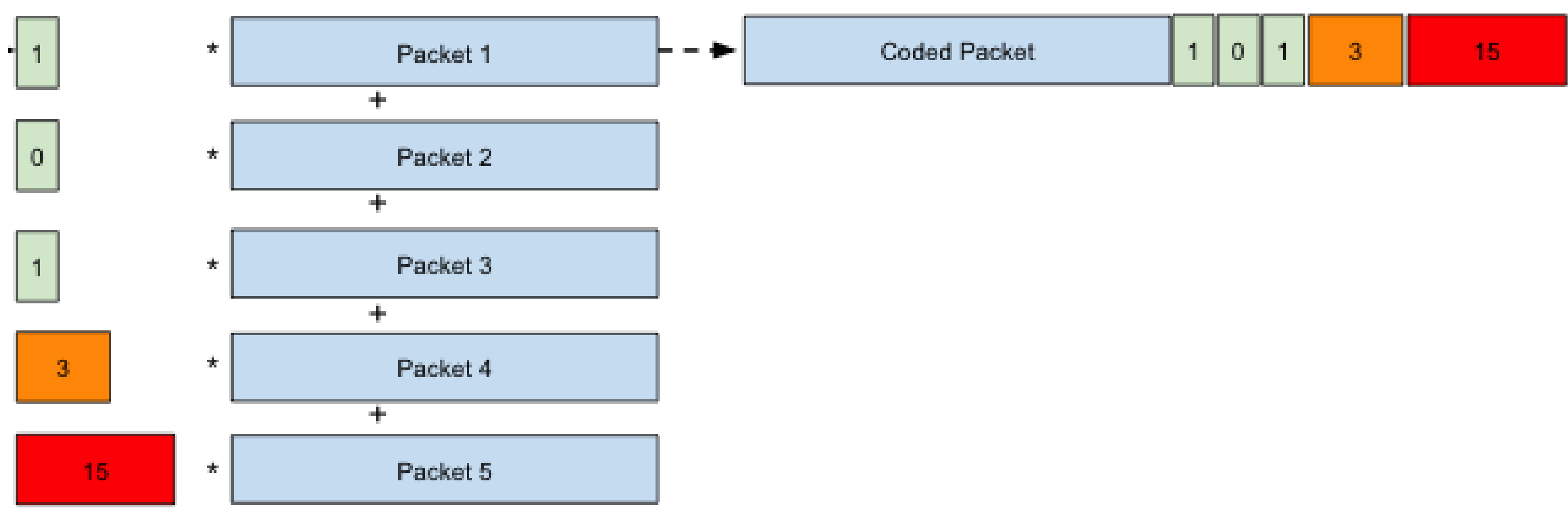
Which results in "5" for the larger field

$GF(2^{2^2})$

x	**1**	**2**	**3**	**4**	**5**	**…**	**15**
1	1	2	3	4	5	…	15
2	2	3	1	8		…	**5**
3	3	1	2	12		…	10
4	4	8	12	6		…	1
5	5					…	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	15	5	10	1		…	9

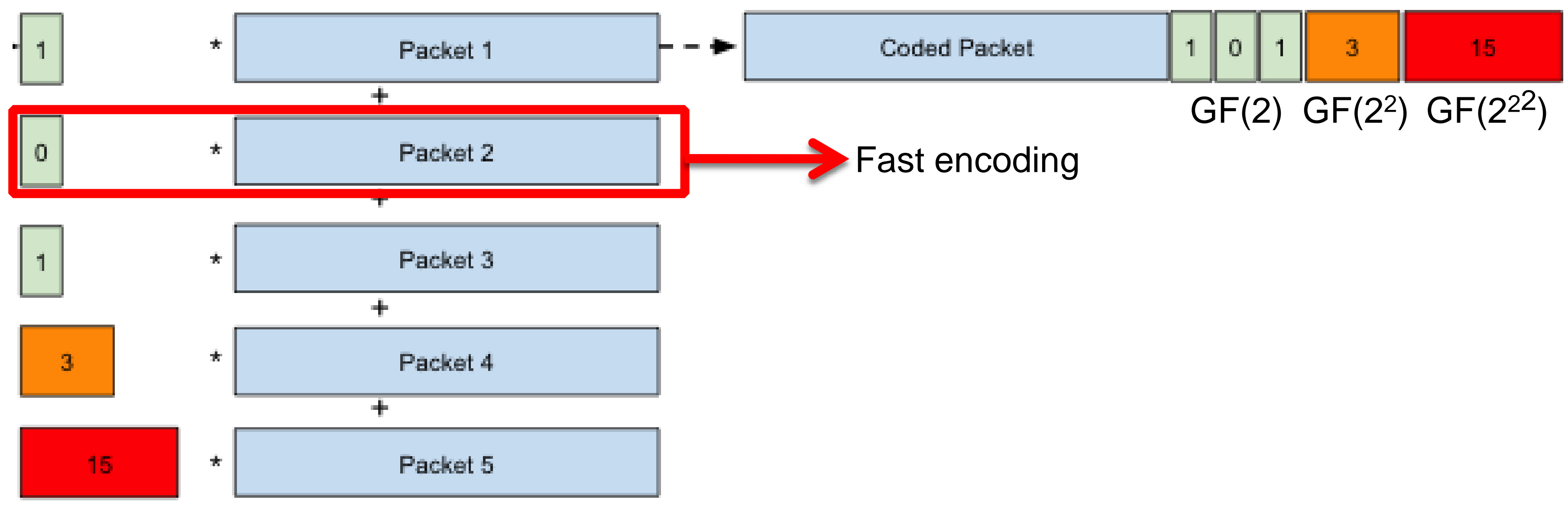
Telescopic Codes

- Design:
  - Multiple composite extension fields
- Goal: reduce overhead, maintaining high performance, faster encoding/decoding
  - Different packets are encoded using different field sizes

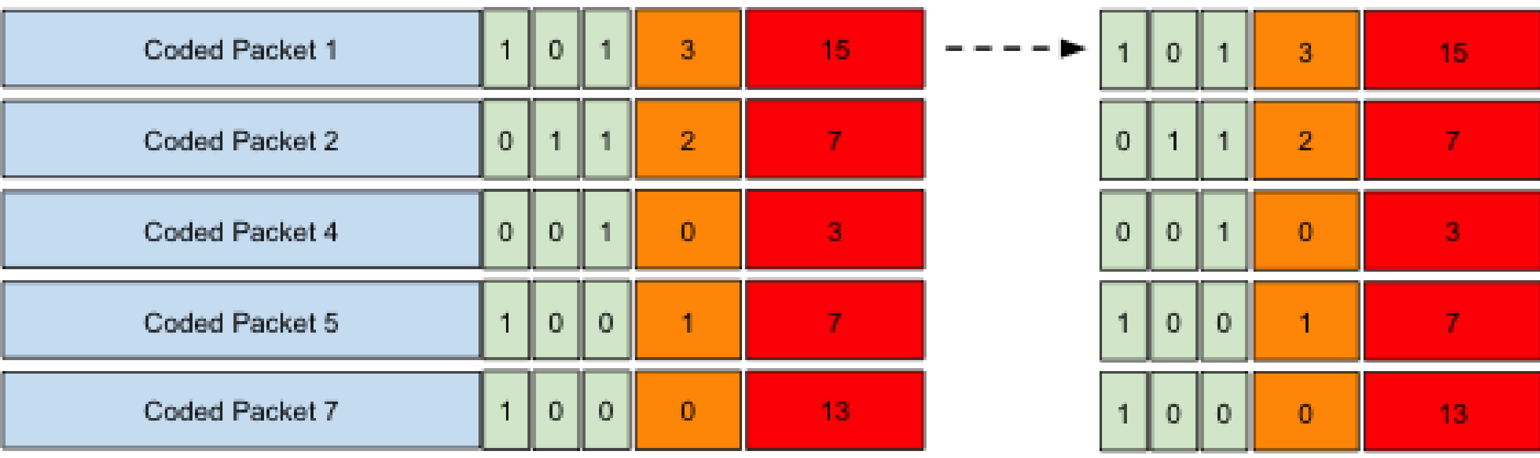


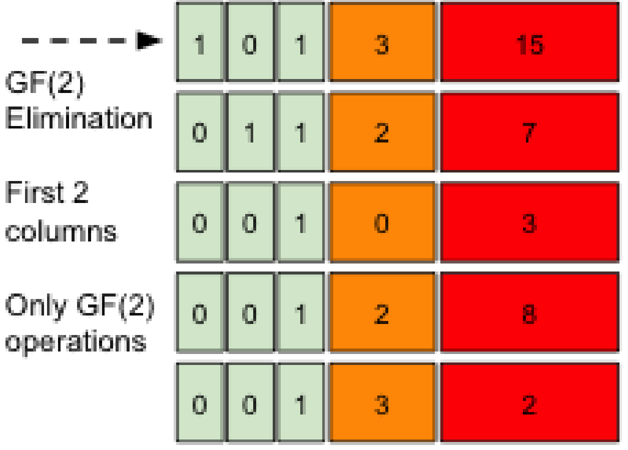
Telescopic Codes

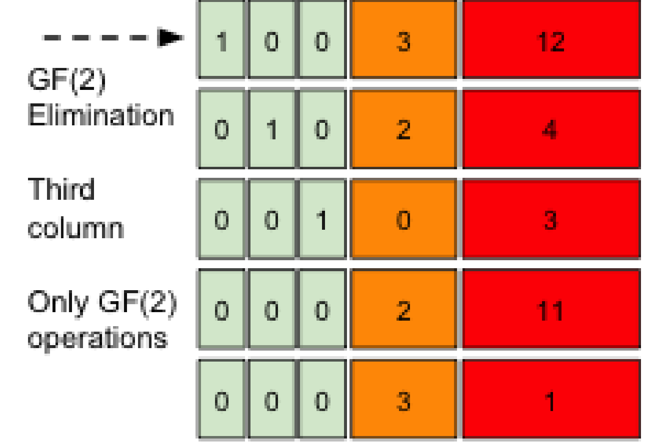
- Design:
  - Multiple composite extension fields
- Goal: reduce overhead, maintaining high performance, faster encoding/decoding
  - Different packets are encoded using different field sizes

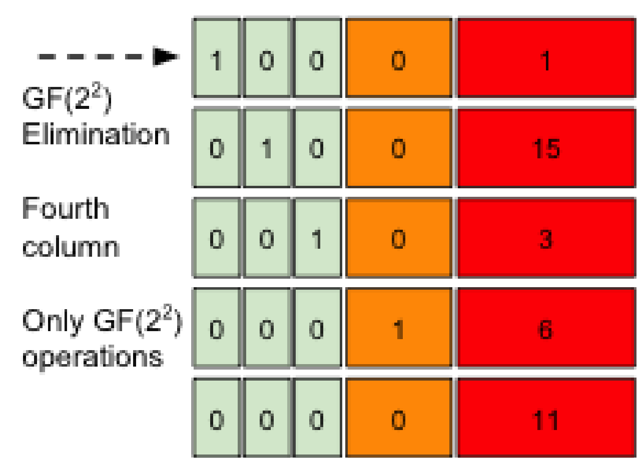


Telescopic Decoder









Telescopic Results

Different choices of q
One-field
Two-fields
Multi-fields
2^1 2^2 2^4 2^8 2^{16} 2^{32}
Overhead compared to the best fixed field
Two-fields
Multi-fields
Total overhead [%]
Overhead reduction [%]
Generation size [-]
4 8 16 32 64 128 256 512

Different choices of q
One-field
Two-fields
Multi-fields
2^1 2^2 2^4 2^8 2^{16} 2^{32}
Overhead compared to the best fixed field
Two-fields
Multi-fields
Total overhead [%]
Overhead reduction [%]
Generation size [-]
4 8 16 32 64 128 256 512 1024 2048 4096 8192

Telescopic Results

Optimal choice of field distribution [%]

100
80
60
40
20
0

4
8
16
32
64
128
256
512

Generation size [-]

Field size
2^{16}
2^{8}
2^{4}
2^{2}
2

Telescopic Results

G=512
If generation size is huge, using binary fields is totally fine as the 1.6 extra packets per generation, here 512, is not really a problem.

Telescopic Results

Optimal choice of field distribution [%]

Generation size [-]

Field size: 2^{16}, 2^8, 2^4, 2^2, 2

G=64
Some coded packets with higher fields in the end to improve the efficiency, but majority of the packets is still coded in the binary field.

Telescopic Results

Optimal choice of field distribution [%]

Generation size [-]

Field size: 2^{16}, 2^8, 2^4, 2^2, 2

G=8
Multi-field is used in order to achieve good efficiency.

Conclusions

- Simple design of composite extension fields
- Proposed telescopic codes
  - Reduce total overhead
  - Maintain high decoding probability
  - Potential for processing at high speed (simpler encoder/decoder)
- Future work
  - Applications in other codes
  - New code designs
  - Performance evaluation in real systems

KODO: Overhead

Janus Heide and Morten V. Pedersen and Frank H.P. Fitzek and Muriel Medard. **On Code Parameters and Coding Vector Representation for Practical RLNC.** 2011. in *IEEE International Conference on Communications (ICC) - Communication Theory Symposium*. Kyoto, Japan.

What is overhead?

- Overhead by encoding vector
  - Generation size G
  - Field size F
  - Payload P
  - Number of additional bits for each packet $A = G * \log_2(F)$ for Full RLNC
  - A can be smaller for Sparse RLNC if encoding vector is compressable
- Linear dependent retransmissions
  - In case a packet is linear dependent it has to be retransmitted (together with the header)
  - $A = P + G * \log_2(F)$

Linear Dependency for Different Density Levels and Field Sizes

$q = 2, d = \frac{1}{2}$
$q = 2, d = \frac{1}{4}$
$q = 2, d = \frac{1}{8}$
$q = 2, d = \frac{1}{16}$
$q = 2^8, d = 1 - 2^{-8}$
$q = 2^8, d = \frac{1}{2}$
$q = 2^8, d = \frac{1}{4}$
$q = 2^8, d = \frac{1}{8}$
$q = 2^8, d = \frac{1}{16}$

Linear dependence overhead [%]
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}

16 32 64 128 256 512
Generation size [-]

Linear Dependency for Different Density Levels and Field Sizes

$q = 2, d = \frac{1}{2}$
$q = 2, d = \frac{1}{4}$
$q = 2, d = \frac{1}{8}$
$q = 2, d = \frac{1}{16}$
$q = 2^8, d = 1 - 2^{-8}$
$q = 2^8, d = \frac{1}{2}$
$q = 2^8, d = \frac{1}{4}$
$q = 2^8, d = \frac{1}{8}$
$q = 2^8, d = \frac{1}{16}$

Linear dependence overhead [%]

10^4 10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3}

16 32 64 128 256 512

Generation size [-]

For the binary field and small generation sizes, the density level has an impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes

$q = 2, d = \frac{1}{2}$
$q = 2, d = \frac{1}{4}$
$q = 2, d = \frac{1}{8}$
$q = 2, d = \frac{1}{16}$
$q = 2^8, d = 1 - 2^{-8}$
$q = 2^8, d = \frac{1}{2}$
$q = 2^8, d = \frac{1}{4}$
$q = 2^8, d = \frac{1}{8}$
$q = 2^8, d = \frac{1}{16}$

Linear dependence overhead [%]
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
16
32
64
128
256
512
Generation size [-]

For the binary field and large generation sizes, the density level has no impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes

$q = 2, d = \frac{1}{2}$
$q = 2, d = \frac{1}{4}$
$q = 2, d = \frac{1}{8}$
$q = 2, d = \frac{1}{16}$
$q = 2^8, d = 1 - 2^{-8}$
$q = 2^8, d = \frac{1}{2}$
$q = 2^8, d = \frac{1}{4}$
$q = 2^8, d = \frac{1}{8}$
$q = 2^8, d = \frac{1}{16}$

Linear dependence overhead [%]

10^{4}, 10^{3}, 10^{2}, 10^{1}, 10^{0}, 10^{-1}, 10^{-2}, 10^{-3}

16 32 64 128 256 512

Generation size [-]

For the binary8 field and small to medium generation sizes, the density level has a huge impact on the overhead.

Linear Dependency for Different Density Levels and Field Sizes

$q = 2, d = \frac{1}{2}$
$q = 2, d = \frac{1}{4}$
$q = 2, d = \frac{1}{8}$
$q = 2, d = \frac{1}{16}$
$q = 2^8, d = 1 - 2^{-8}$
$q = 2^8, d = \frac{1}{2}$
$q = 2^8, d = \frac{1}{4}$
$q = 2^8, d = \frac{1}{8}$
$q = 2^8, d = \frac{1}{16}$

Linear dependence overhead [%]
10^4 10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3}
16 32 64 128 256 512
Generation size [-]

For the binary8 field and very large generation sizes, the density level has no impact on the overhead.

So where does the overhead comes from?

Linear Dependency for Different Density Levels

Designing rules …

(a) $q = 2,\ g = 64$

(b) $q = 2,\ g = 1024$

(c) $q = 2^8,\ g = 1024$

α = additonal transmissions due to field size

β = additional packets due to linear dependency

γ = bits in the encoding vector

Linear Dependency for Different Density Levels

F=2;G=64

Linear Dependency for Different Density Levels

F=2;G=1024

Total overhead [%]

10^{2}, 10^{1}, 10^{0}, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}

β = additional packets due to linear dependency

γ = bits in the encoding vector

α = additonal transmissions due to field size

10^{-3}, 10^{-2}, 10^{-1}, 10^{0}

Density [-]

Linear Dependency for Different Density Levels

$F=2^8;G=1024$

Total overhead [%]

10^{2}, 10^{1}, 10^{0}, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}

β

γ

β = additional packets due to linear dependency

γ = bits in the encoding vector

α = additonal transmissions due to field size

10^{-3}, 10^{-2}, 10^{-1}, 10^{0}

Density [-]

Linear Dependency for Different Density Levels

F=2⁸;G=1024

Risk of additional linear dependency mitigates for more dense encoding vectors.

Number of bits in encoding vector changes with the density level due to compression.

No dependency between density level and field size.

Total overhead

Density [-]

10^{2} 10^{1} 10^{0} 10^{-1} 10^{-2} 10^{-3} 10^{-4} 10^{-5} 10^{-6}

10^{-3} 10^{-2} 10^{-1} 10^{0}

β γ

β = additional packets due to linear dependency

γ = bits in the encoding vector

α = additonal transmissions due to field size

Linear Dependency for Different Density Levels

optimal density level

β = additional packets due to linear dependency

γ = bits in the encoding vector

α = additonal transmissions due to field size

Total overhead [%]

Density [-]

Effects of Heterogeneous Packet lengths on Network Coding

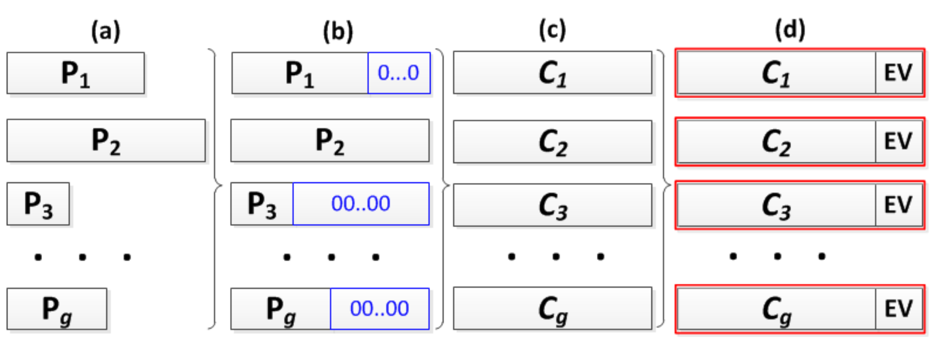
Effects of Heterogeneous Packet lengths on Network Coding

Overhead due to the RLNC encoding process

(a) Coding overhead: defined by the Encoding Vector size, containing the coding coefficients.

(b) Linear dependency overhead: need of extra transmissions when linear dependent combinations are received

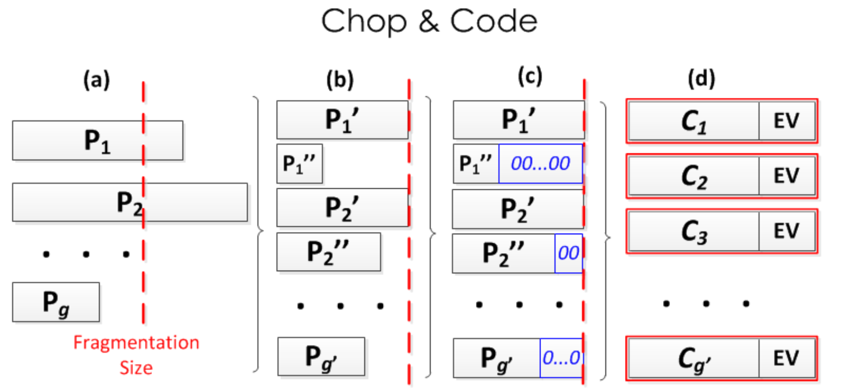
(c) Padding overhead: created by coding with the biggest data packet of a generation with different sizes

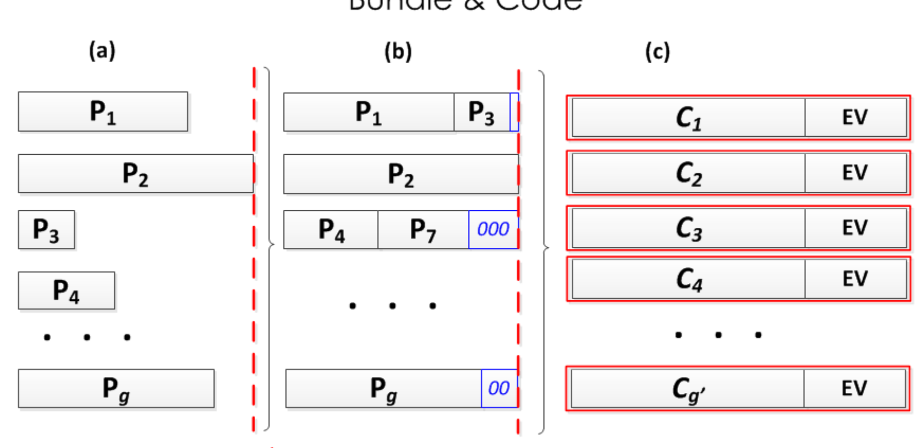


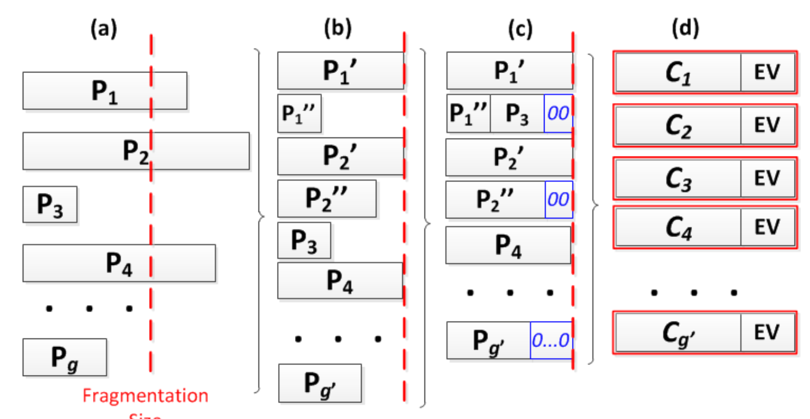
Naive assumption: all packets in a generation have the same size

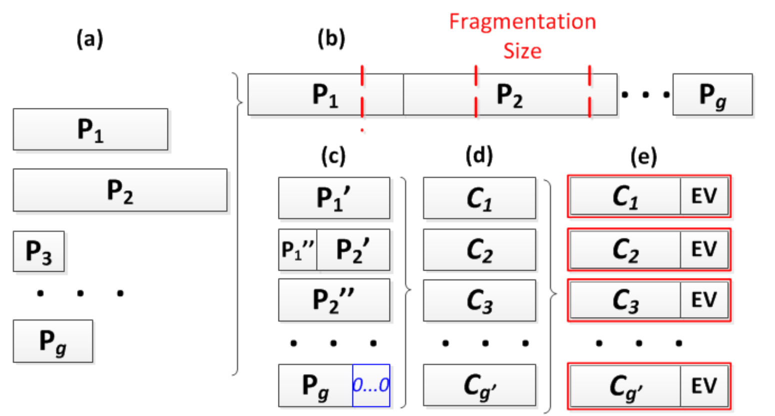
Real life data: heterogenousity of packet lengths

Effects of Heterogeneous Packet lengths on Network Coding



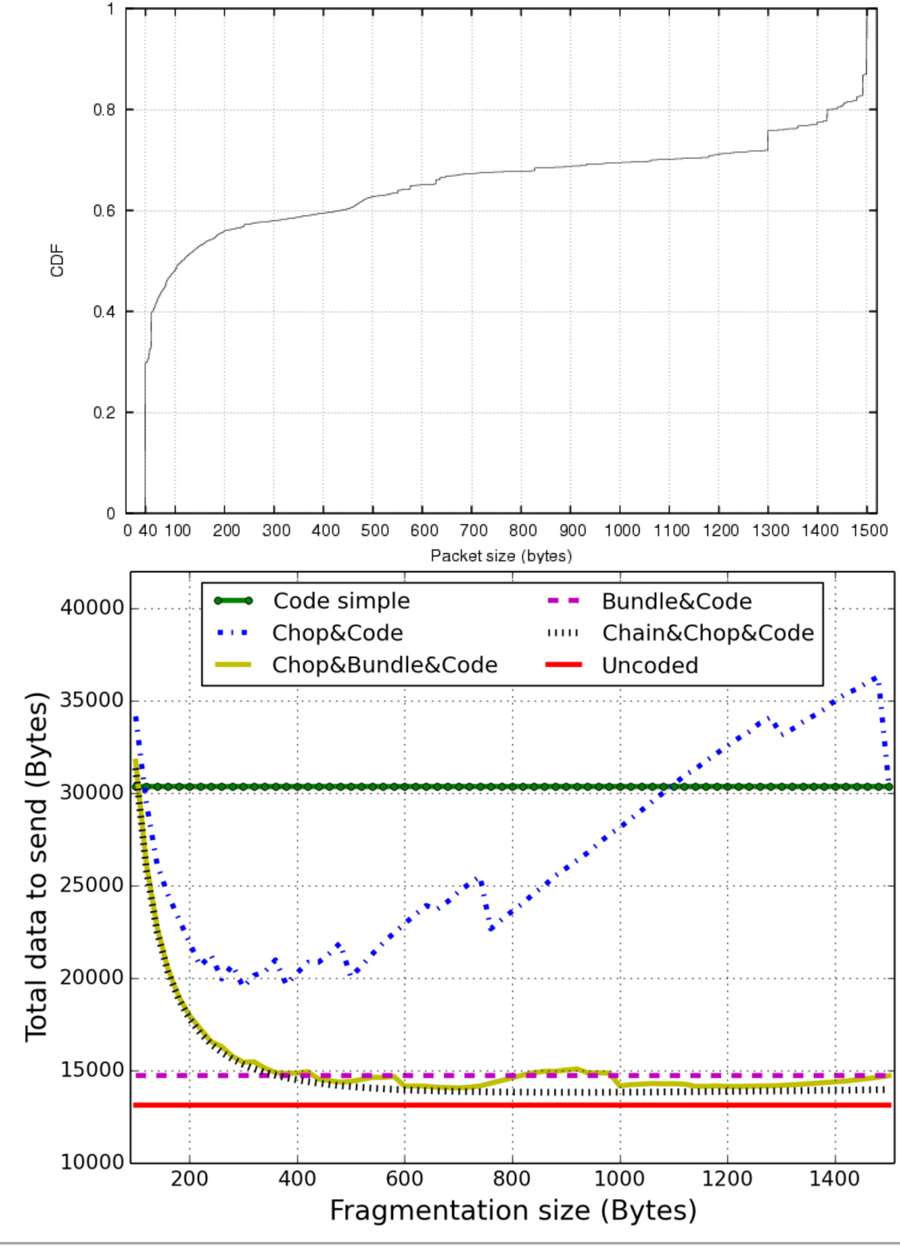






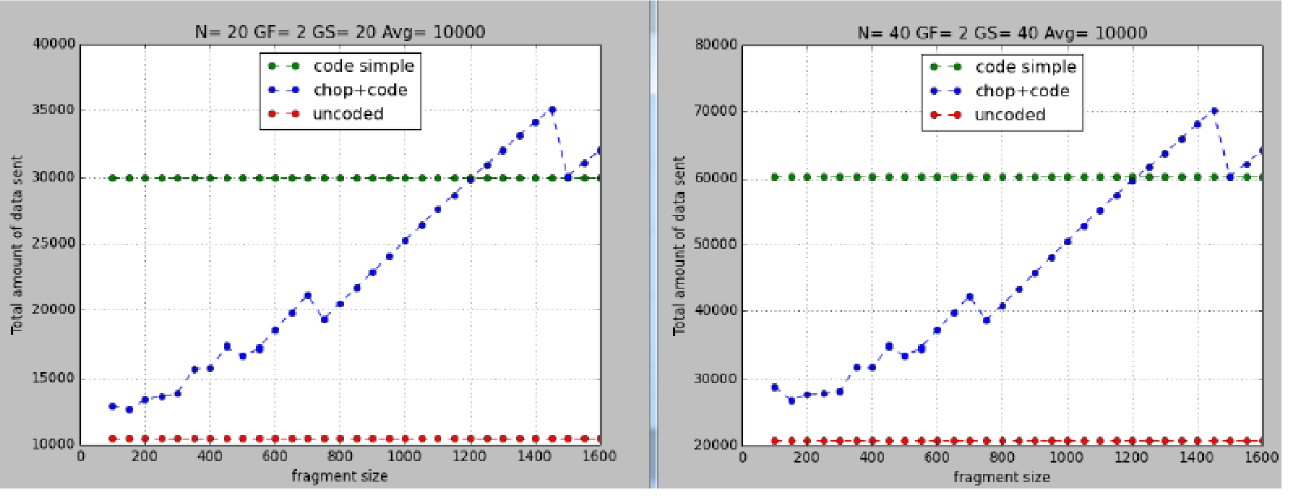
Internet data from CAIDA

- Generations of 20 packets with field size $q=2^8$
- Packets are chosen randomly following the CDF, and run the simulations 10000 times and averaged the results
- Internet packet size distribution is mostly of 40 Bytes and 1500 Bytes (with approximate probabilities of 40% and 20% respectively)[1]

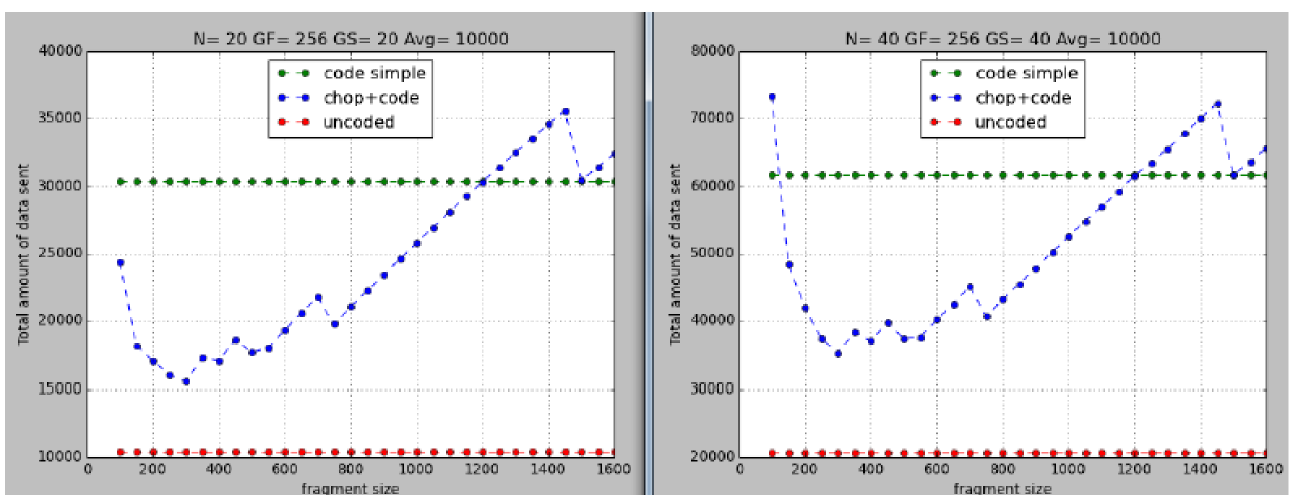


[1] http://www.isi.edu/~johnh/PAPERS/Sinha07a.pdf

Packet Size Comparison 1

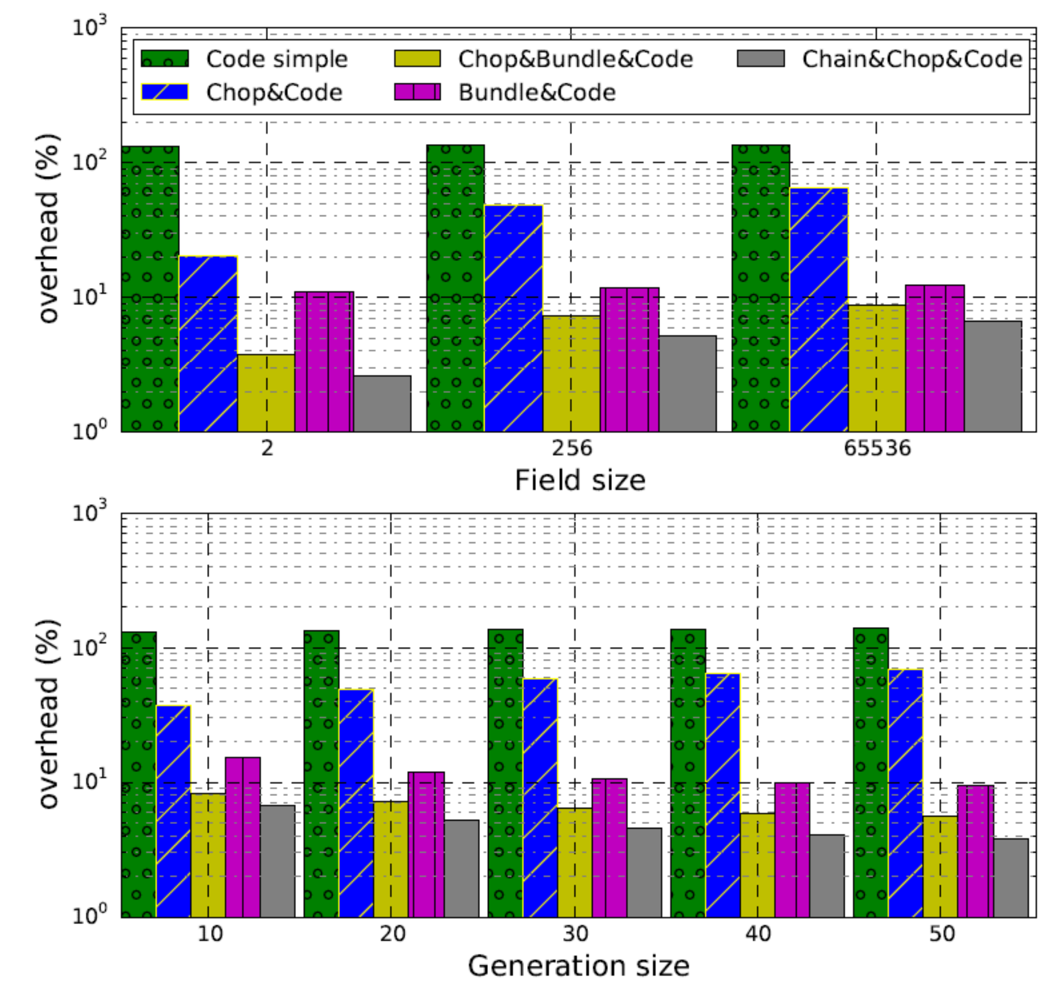


GF=2^8



Internet data from CAIDA (2)

- Total coding overhead (%) added for each solution respect the uncoded data for:
  - different field sizes (and generation size constant of 20 packets)
  - different generation sizes (and field size constant $q=2^8$)
- Coding directly adds more than 100% of overhead.
- Chain& Chop& Code achieves to reduce to less than 5% for a large generation size.



Packet Size Example 2

NAL length (Bytes)
0 20000 40000 60000 80000 100000 120000
NAL index
0 50 100 150 200 250 300 350

Packet Size Comparison 2

N= 25 GF= 2 GS= 25 Window size= 25 Total data length= 322

code simple
Optimal value chop+code
uncoded

Total amount of data

2600000
2400000
2200000
2000000
1800000
1600000
1400000
1200000
1000000

0 50 100 150 200 250 300

iterations

KODO: Energy consumption and complexity

Energy consumption: KODO

Energy consumption: KODO

TABLE I
OVERVIEW OF THE FINITE FIELD IMPLEMENTATIONS USED.

$GF(2)$	The binary field $GF(2)$ was implemented using only the XOR operation, which typically yields very efficient implementations on most modern processors.
$GF(2^8)$	The binary extension field $GF(2^8)$ uses full multiplication and division lookup tables. Using this approach will replace the required polynomial multiplication or division of two field elements with a single table lookup. More details on these lookup tables are described in [17]. Addition and subtraction require only a bit-by-bit XOR.
$GF(2^{16})$	Computing a full multiplication and division table for $GF(2^{16})$ will typically require too much memory for most platforms (in the order of GBs). Instead, our implementation uses an extended logarithmic lookup, which reduces the memory requirements to the order of KBs. Addition and subtraction are performed as bit-by-bit XOR operation. Details are provided in [17].
$GF(2^{32}-5)$	This field is also known as an OPF (Optimal Prime Field). It has the advantage that it does not require any lookup tables and uses the standard arithmetic logic unit of the processor for all arithmetic operations. A detailed description of this field can be found in [18].

Measurement Setup

Settings
Field
Generation

Data → Kodo → WiFi

Mobile Phone

Settings
Field
Generation

WiFi ← Kodo ← Data

Mobile Phone

Energy consumption: KODO

Power [W]

2.5
2
1.5
1
0.5
0

Ace
Sony Xperia
S3

GF (2)
GF (2^8)
GF (2^{16})
GF (2^{32}-5)
Other Processes and Components
WiFi Tx.
WiFi Rx.

Energy consumption: KODO

TABLE I
ENCODE AND DECODE PROCESSING SPEED (MBPS) AND AVERAGE PROCESSING POWER (W) FOR THREE MOBILE DEVICES.

Encoder	$GF(2)$			$GF(2^8)$			$GF(2^{16})$			$GF(2^{32}-5)$		
Pkts/Gen.	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3
2	1675.8	3328.1	3916.22	238.4	463.2	669.71	45.5	100.5	216.67	175.6	293.1	396.52
4	921.1	1615.3	2413.61	120.1	233.0	342.34	18.9	46.2	99.14	89.1	148.0	209.18
8	527.9	938.2	1394.60	59.4	117.4	172.91	9.1	24.8	36.04	44.9	74.0	80.76
16	253.1	347.1	640.2	30.5	58.3	85.6	4.7	14.1	27.1	22.7	36.7	52.5
32	132.6	206.9	338.6	15.3	28.5	41.6	2.4	6.2	13.9	11.3	18.0	25.8
64	66.5	99.8	152.3	7.5	14.3	20.7	1.2	3.1	7.0	5.6	9.0	12.8
128	30.3	49.5	77.6	3.6	7.1	10.3	0.6	1.6	3.5	2.8	4.5	6.4
256	11.7	23.3	38.7	1.6	3.4	5.1	0.3	0.8	1.6	1.3	2.2	3.2
512	4.8	10.8	18.5	0.8	1.7	2.5	0.1	0.4	0.8	0.6	1.1	1.6
Decoder	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3	Ace	Xperia	S3
2	2079.0	4005.0	5545.00	148.6	345.3	477.57	53.5	120.7	203.53	187.6	325.5	440.20
4	944.6	1774.0	2499.21	92.2	200.7	294.45	17.9	50.8	102.71	96.8	166.1	231.63
8	561.2	998.8	1384.16	51.9	108.9	158.78	8.7	24.4	45.01	51.1	82.2	103.40
16	287.0	527.3	783.7	27.7	56.2	82.7	5.1	13.1	25.7	25.0	39.9	56.6
32	150.6	256.2	403.2	14.4	27.8	40.3	2.5	7.3	13.3	12.0	18.7	26.7
64	67.6	107.8	169.6	7.3	13.8	20.0	1.2	3.5	6.9	5.6	8.6	12.2
128	31.9	49.8	77.3	3.4	6.6	9.6	0.6	1.6	3.4	2.4	3.7	5.4
256	12.7	24.4	38.4	1.4	3.0	4.5	0.3	0.7	1.4	0.9	1.5	2.1
512	4.8	10.7	18.1	0.6	1.3	1.9	0.1	0.3	0.6	0.3	0.5	0.8
Power (W)	0.486	0.474	1.012	0.421	0.377	1.057	0.279	0.378	1.041	0.425	0.448	1.090

Energy consumption: KODO

Energy [µJ/b]

10
1
0.1
0.01
0.001
0.0001

1 Mbps

8.8 Mbps

2 4 8 16 32 64 128 256 512

Generation size

- GF(2) Decoder
- $GF(2^8)$ Decoder
- $GF(2^{16})$ Decoder
- $GF(2^{32}-5)$ Decoder
- Tx Cost for GF(2)
- Tx Cost for $GF(2^8)$
- Tx Cost for $GF(2^{16})$
- Tx Cost for $GF(2^{32}-5)$

Energy consumption: KODO

Tx and Encoding/Decoding Energy Ratio

10,000
1,000
100
10
1
0.1

GF(2) Encoder
GF(2^8) Encoder
GF(2^{16}) Encoder
GF(2^{32}–5) Encoder
GF(2) Decoder
GF(2^8) Decoder
GF(2^{16}) Decoder
GF(2^{32}–5) Decoder

2 4 8 16 32 64 128 256 512

Generation size

Energy consumption: KODO

$g(1+\alpha)/R_C$

$g(1+\alpha)/R_{TX}$

g/R_D

Transmitter

P_C

Encode Energy

P_{Tx}

Transmit Energy

time

Receivers

P_{Rx}

Receive Energy Rx 1

⋮

P_{Rx}

Receive Energy Rx N

P_D

Decode Energy Rx 1

⋮

P_D

Decode Energy Rx N

Energy consumption: KODO

Energy savings gain

4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

2 4 8 16 32 64 128 256 512

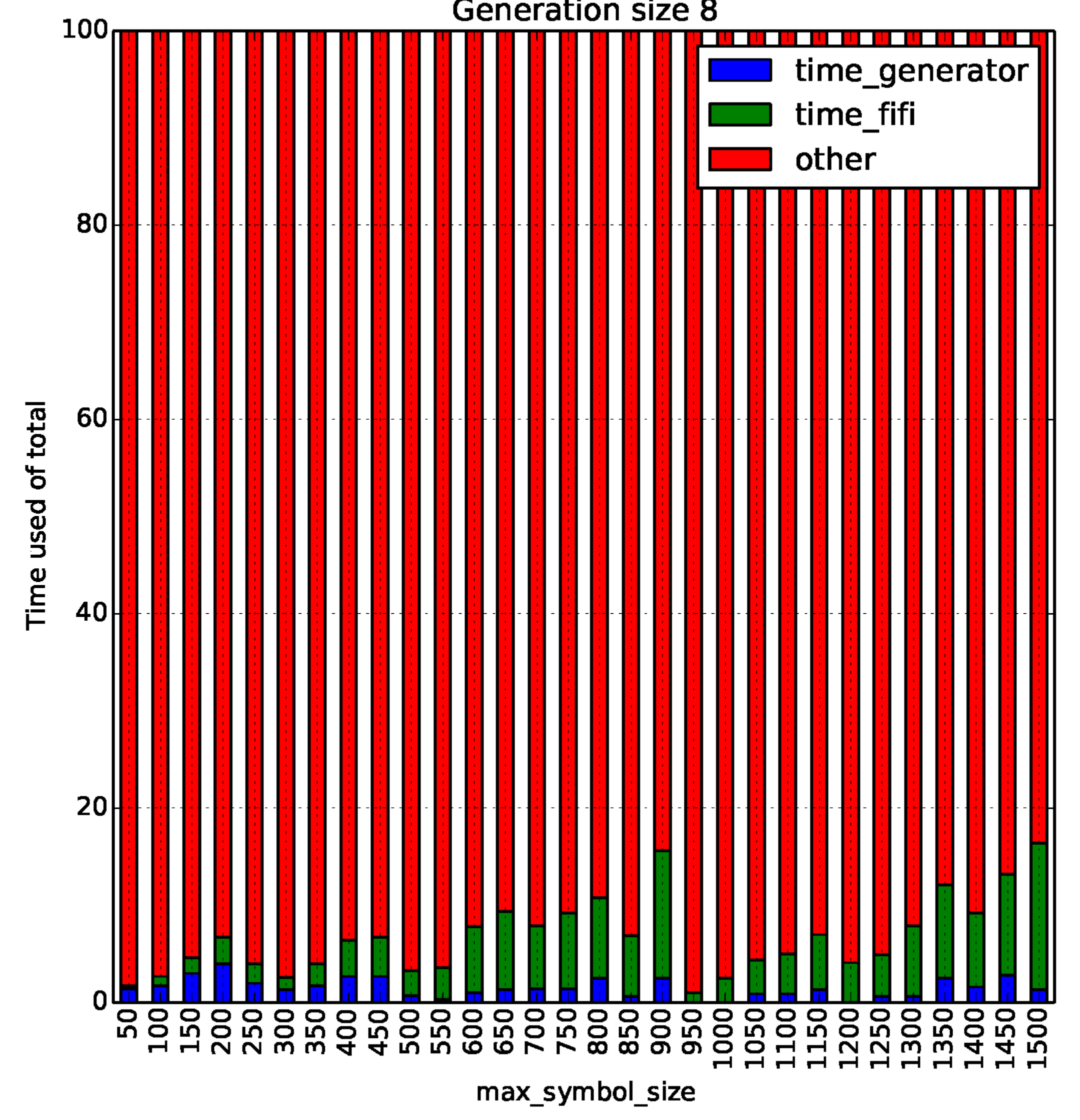
Generation size (packets)

GF(2) lower bound
GF(2^8) lower bound
GF(2^{16}) lower bound
GF(2^{32} – 5) lower bound

KODO: Practical Implementation

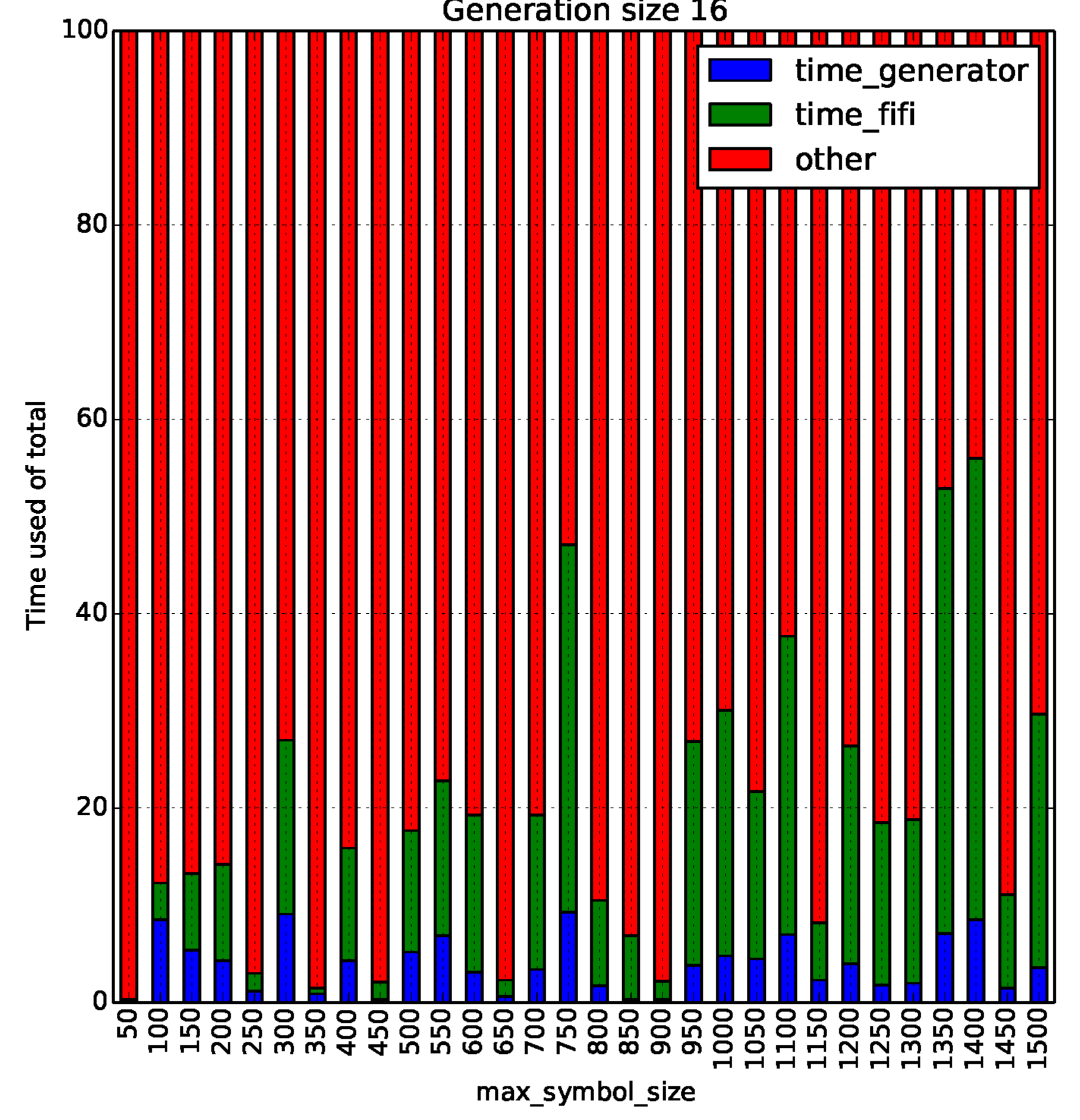
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



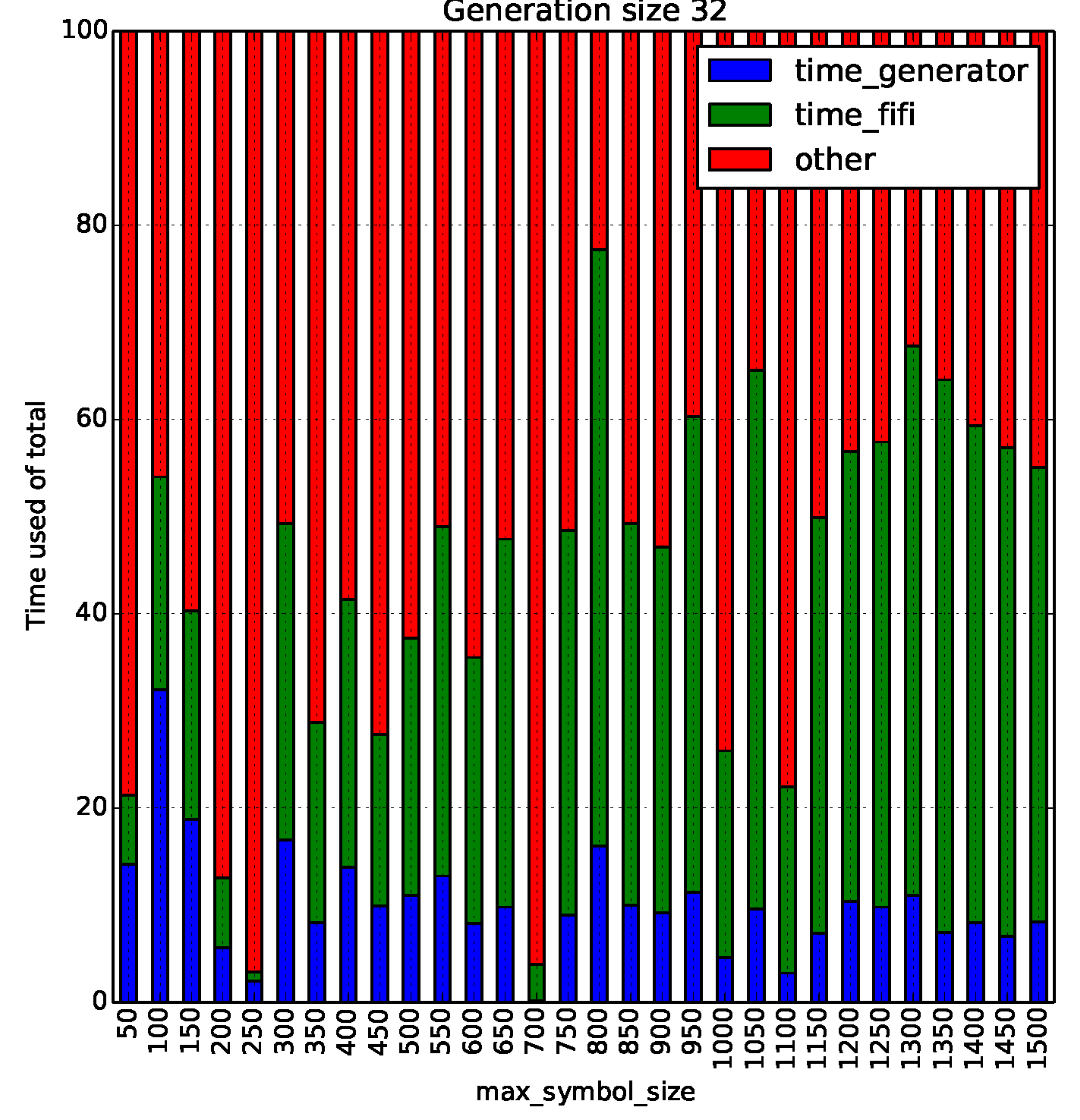
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



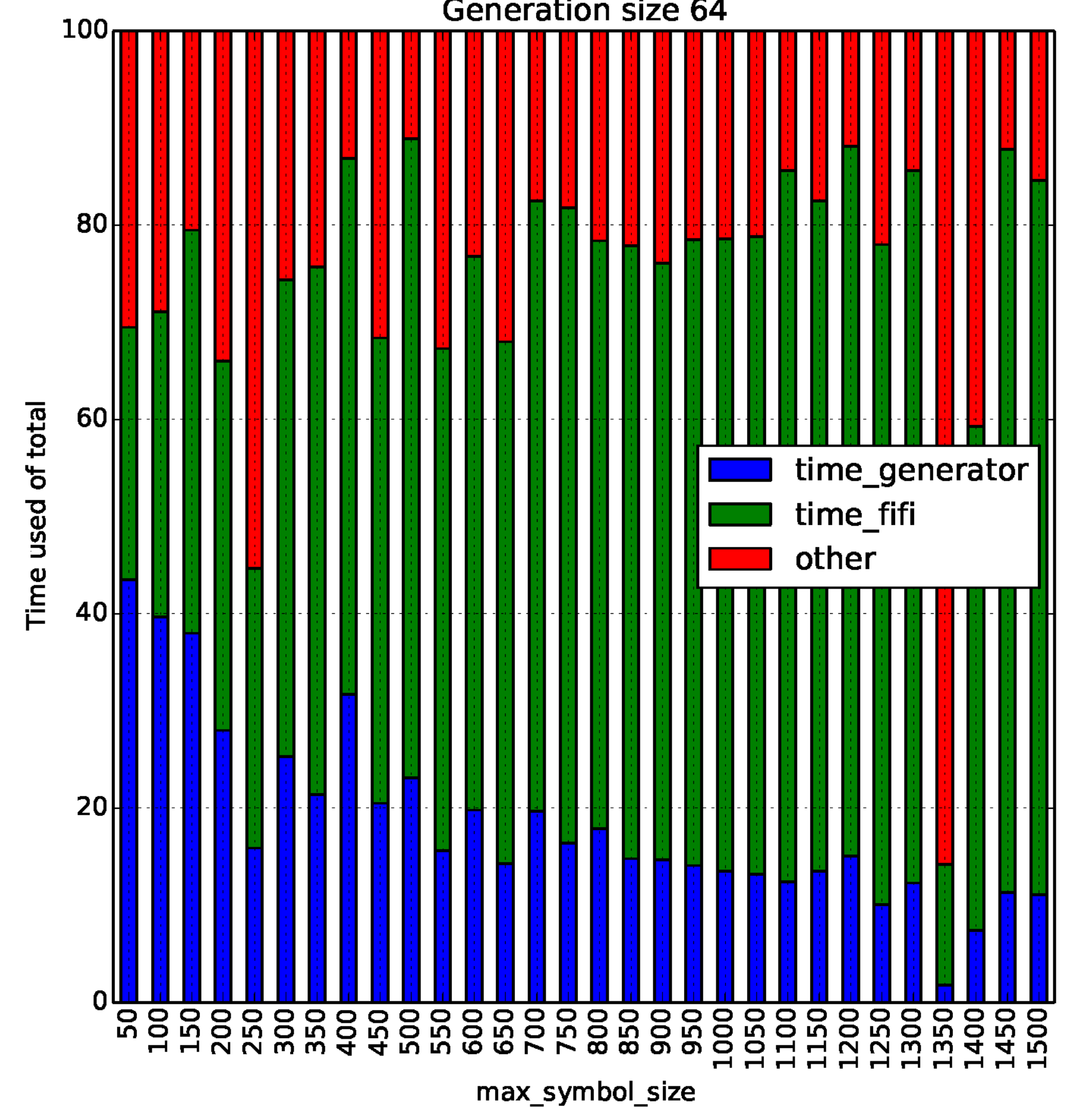
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



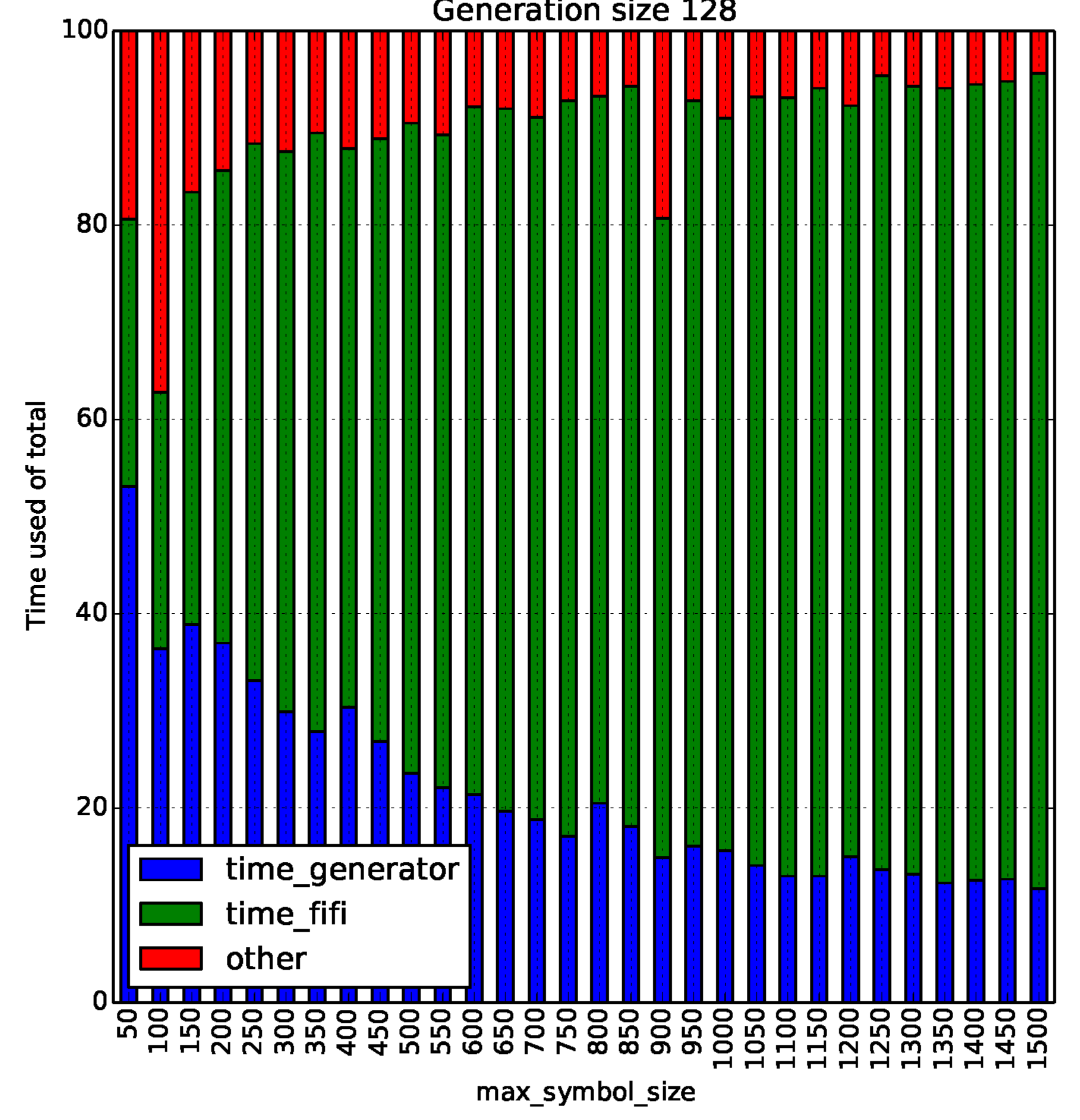
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



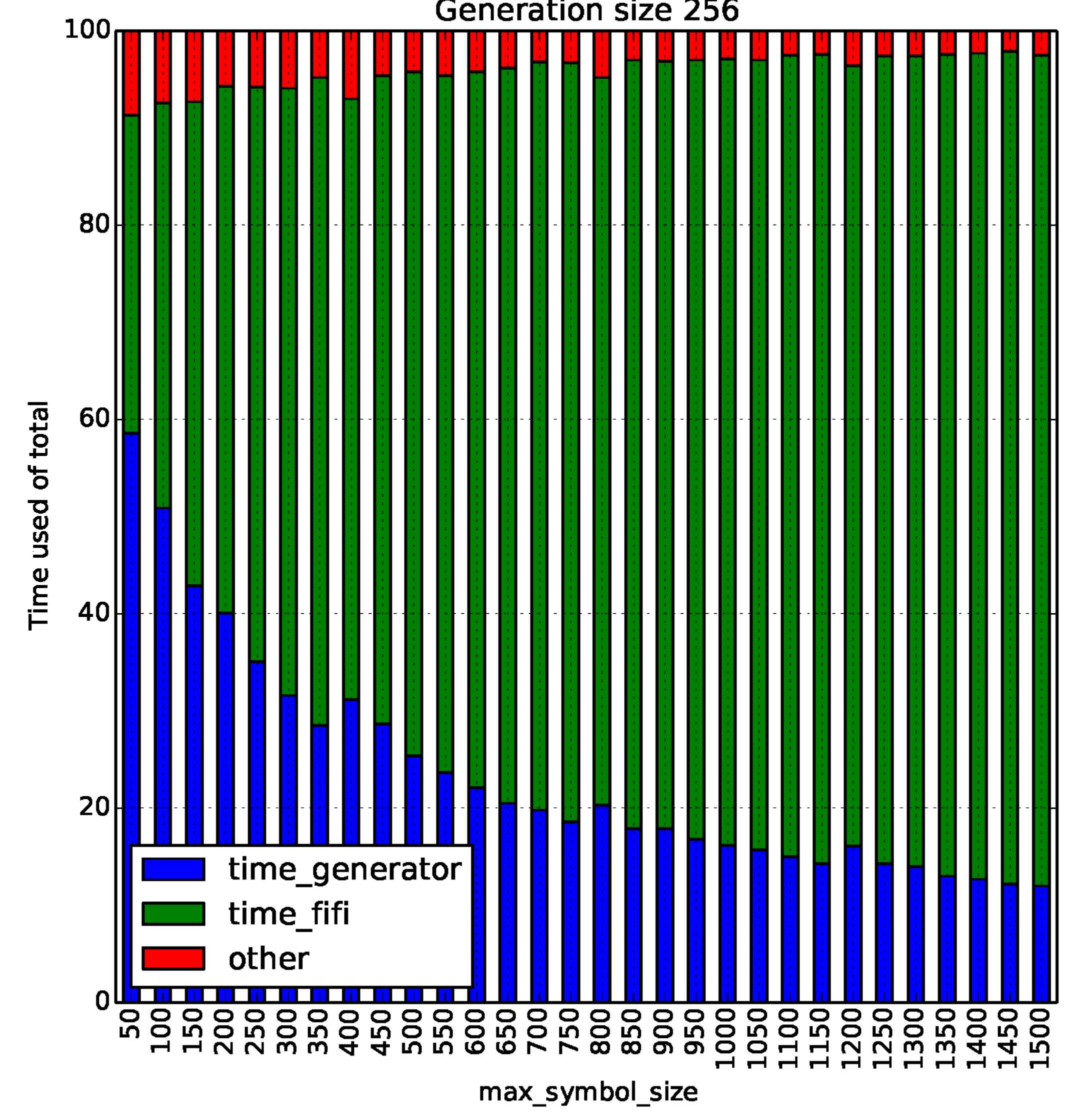
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



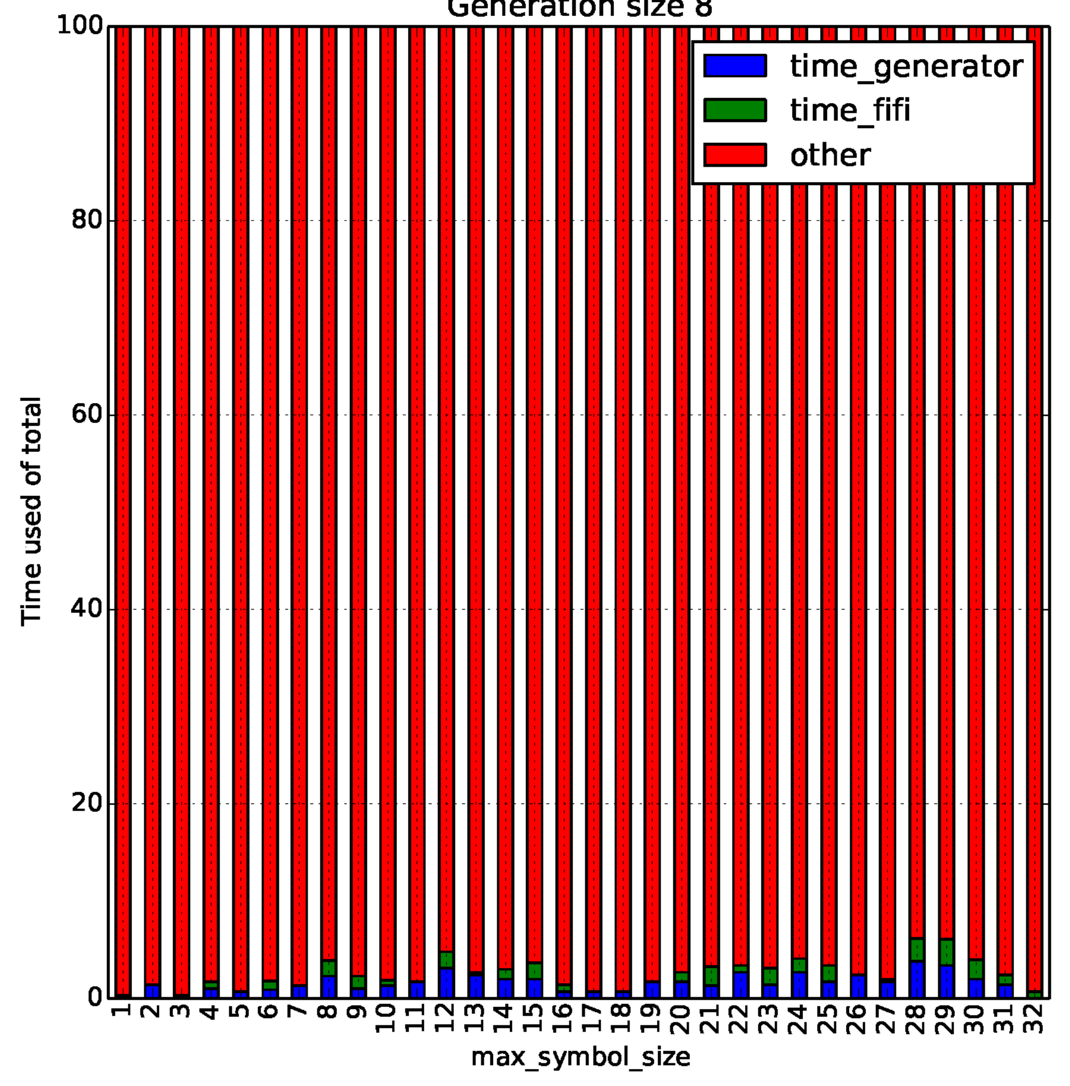
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo



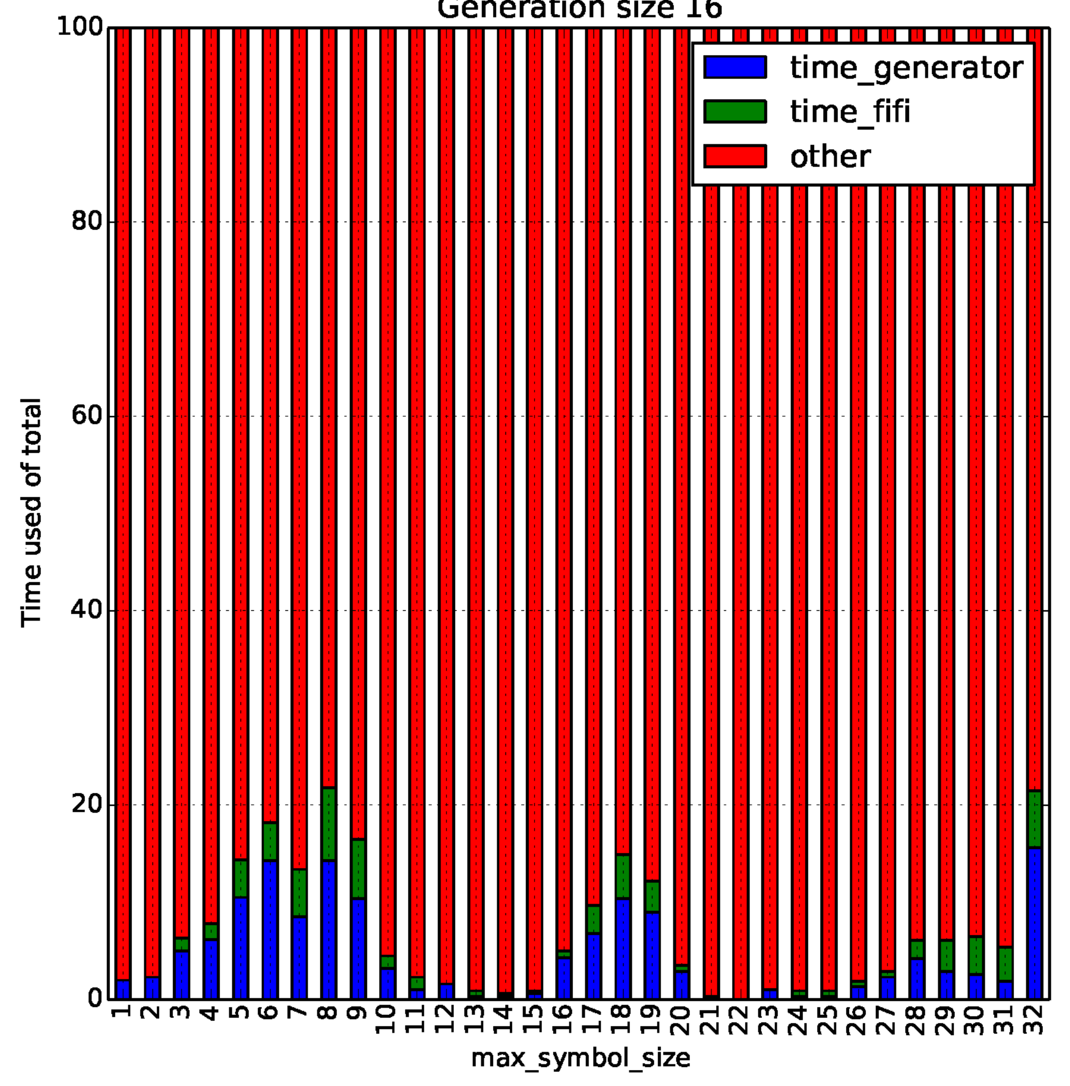
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



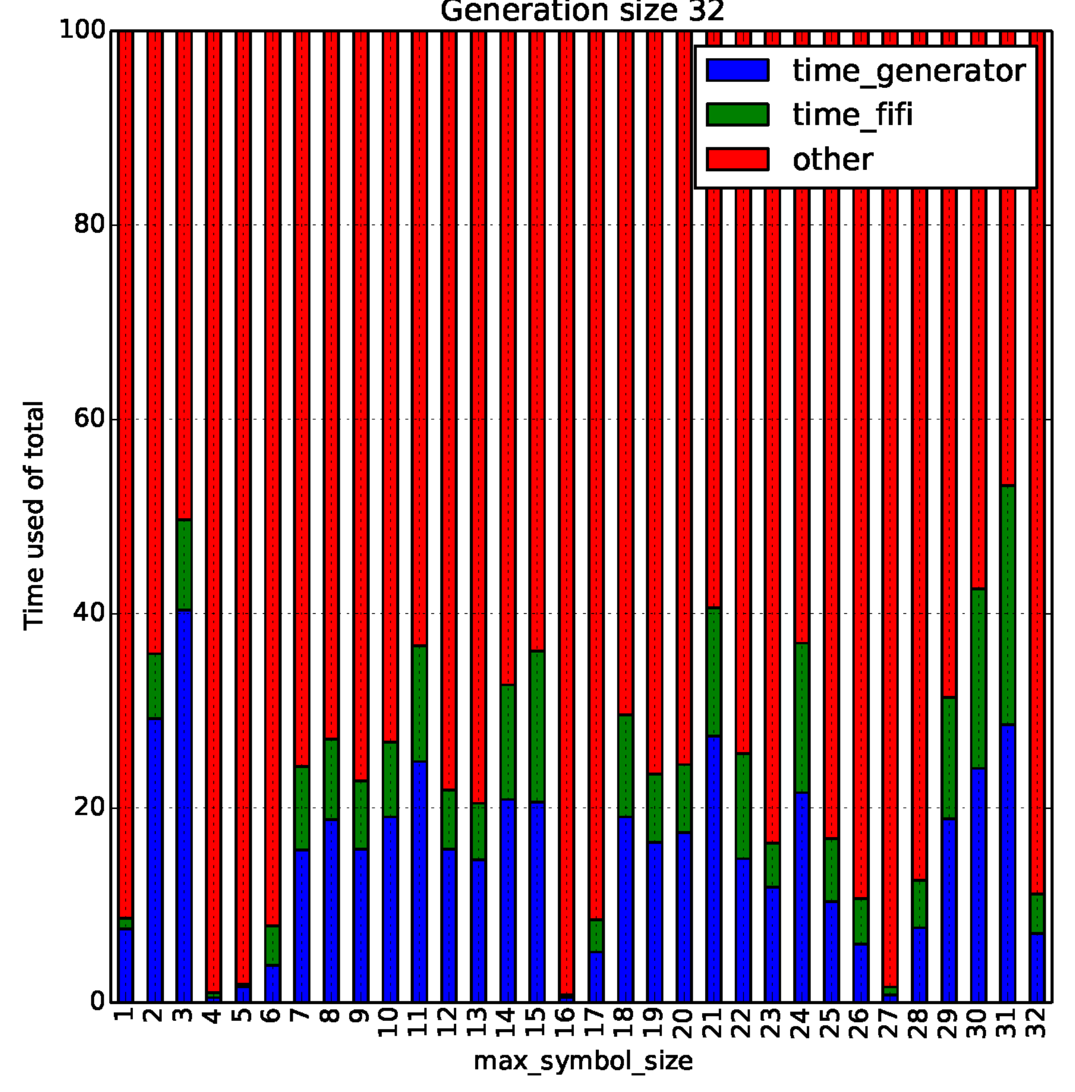
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



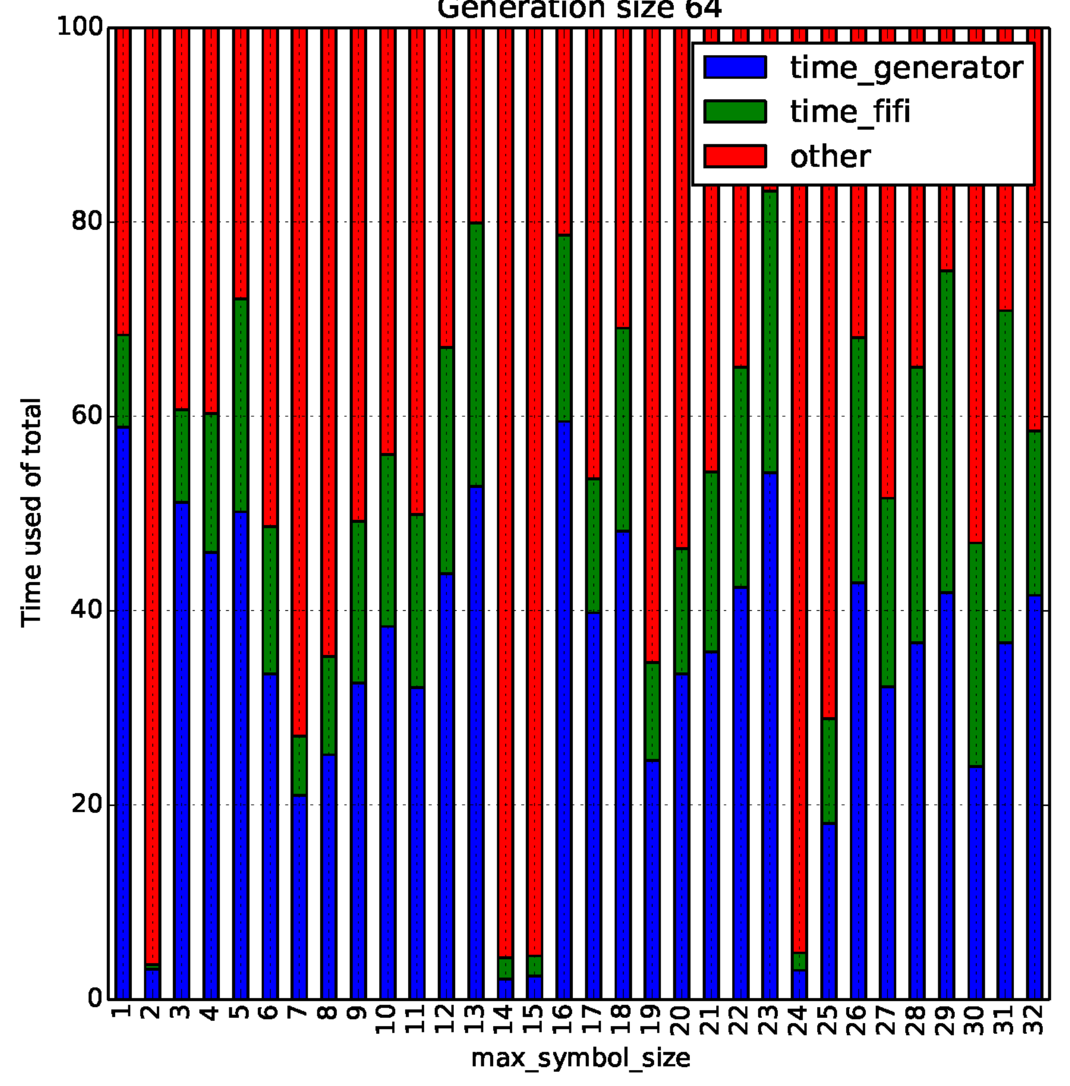
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



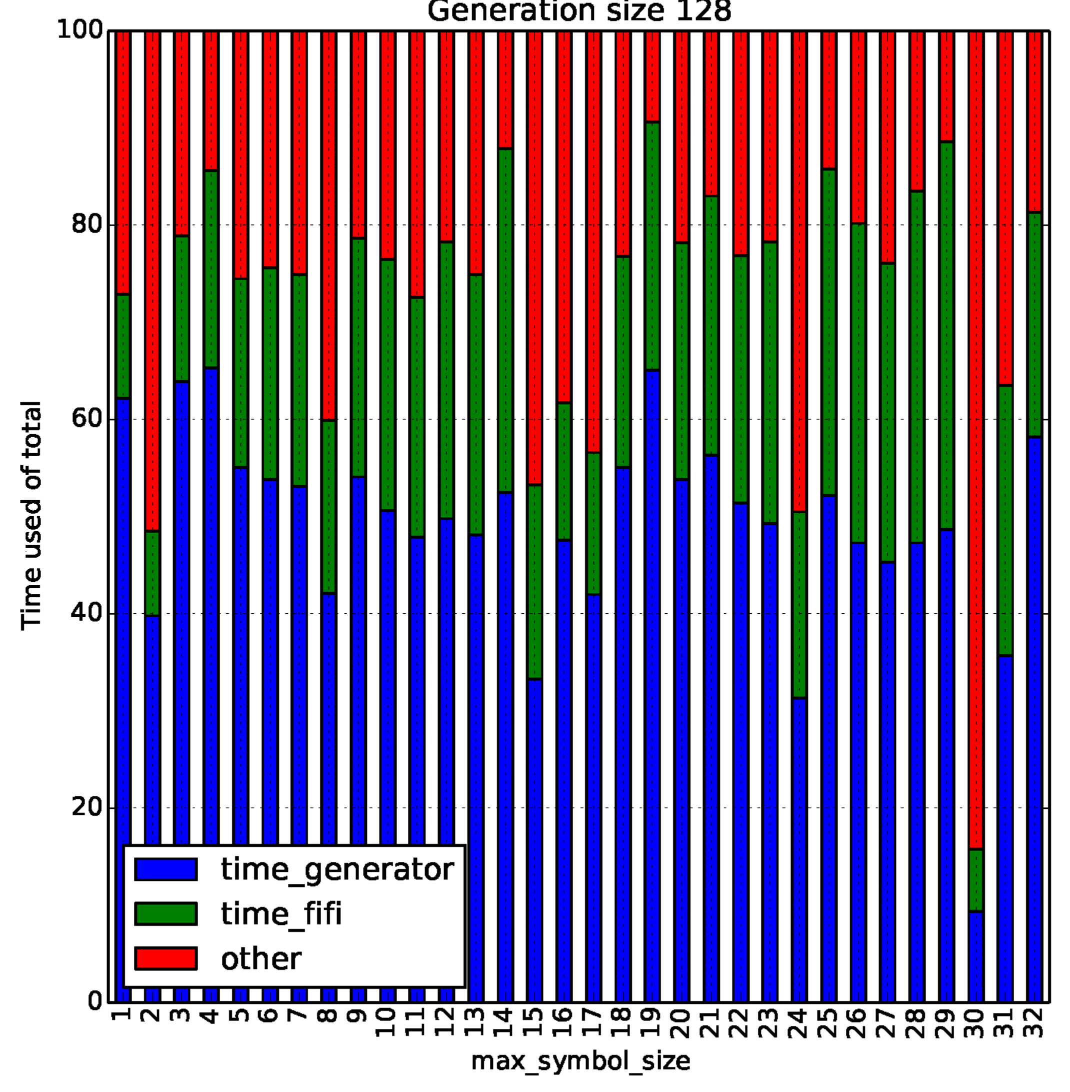
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



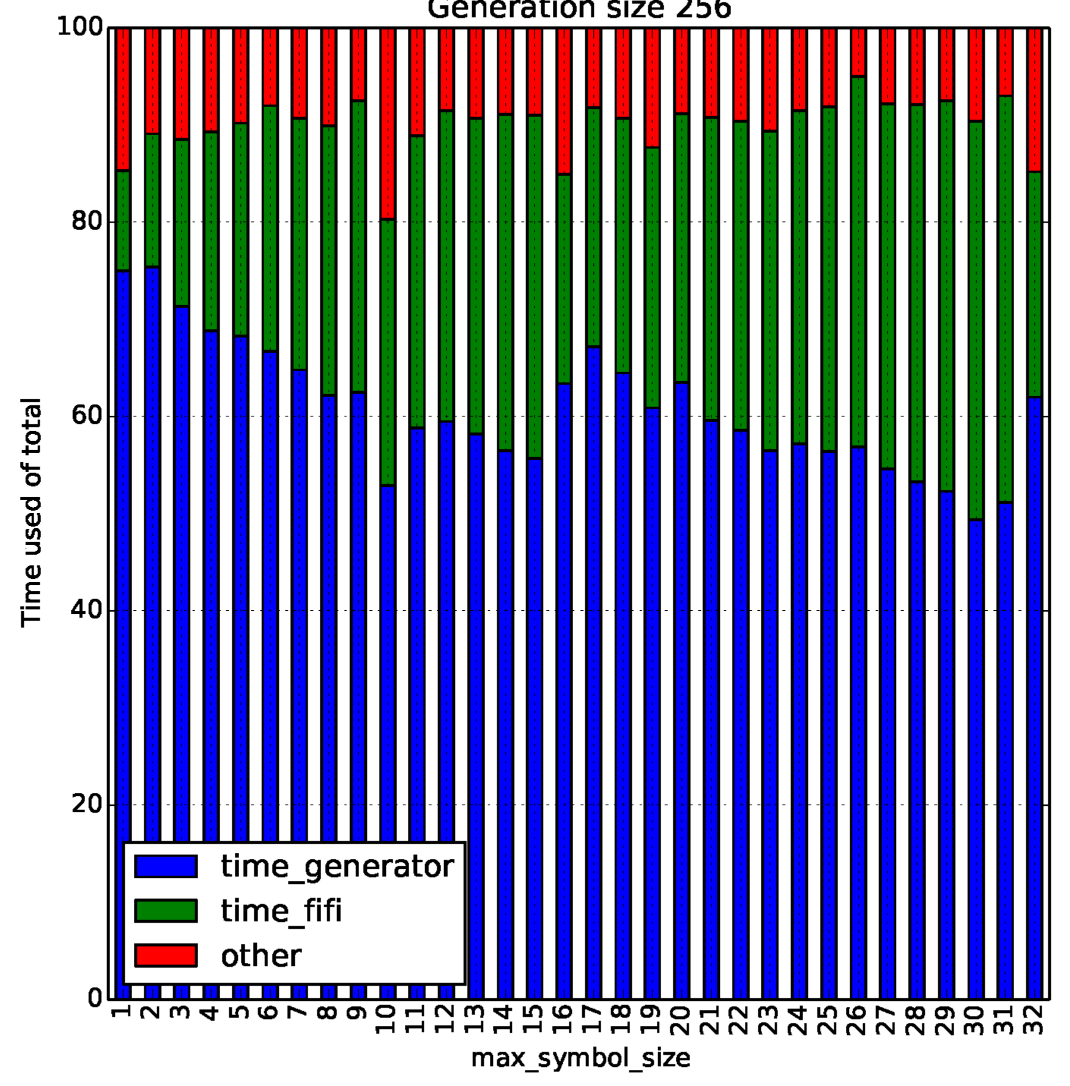
Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



Time Consumed for Coding Activities

- Time for the random generator
- Time for the FIFI calculation
- Rest like updating state in the decoder etc. , basically all other functions in Kodo
- SMALL packets, e.g. control



Many-Core Implementation of Network Coding

base
encoding base
decoding scheduled 1 thread
decoding scheduled 2 thread
decoding scheduled 4 thread
encoding scheduled 1 thread
encoding scheduled 2 thread
encoding scheduled 4 thread

raw throughput [MByte/s]

0.0764 0.2272 0.1261 0.2319 0.3836 0.2229 0.4333 0.8236

Test cases

Herb Sutter: „Free lunch is over!“

Lots of research in computer science and engineering focus on achieving low computational complexity (the big O). But perhaps in the future we need to consider algorithms with worse computational complexity but which are easy to parallelize.

On Raspberry Pi 2: 10x speed up over standard SIMD encoding by using 4 cores and cache optimization (generation size 1024)

Random Number Generator

(and why it matters)

Random Number Generator Introduction

TOUR OF ACCOUNTING

OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.

NINE NINE NINE NINE NINE NINE

ARE YOU SURE THAT'S RANDOM?

THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.

www.dilbert.com scottadams@aol.com

10/25/01 © 2001 United Feature Syndicate, Inc.

Random Number Generator

Random number generators applications

- computer simulation / cryptography / gaming / sampling and coding

Deterministic vs Non-Deterministic

- ND: physical phenomenon that is expected to be random
  - measuring atmospheric noise
  - thermal noise
  - external electromagnetic (AM/FM)
  - quantum phenomena
- D: apparently random results, which are in fact completely deterministic In a given interval by a shorter initial value → pseudo-random number generators
- Random number generation from a probability distribution

Properties of Random Number Generators

Assume sequence of random number R_1,R_2,R_3,...

What are good random numbers?

Two properties:

- *Uniformity*
  - — If the interval [0,1) with N observations is divided into n subintervals of equal length, the number of observation is n/N
- *Independence*
  - — The probability of observing one value in a particular subinterval is independent of previous or future values.

Properties of Random Number Generator

Fast: Simulations and others need a larger number of random numbers

Portability: The generator should be portable to different programming languages to test on different platforms

Long cycle: In order to have a lot of variety, the cycle should not be small

Repeatability: In order to test different approaches the SAME set of random numbers should be used receive comparable results.

Ideal: Uniformity and Independence

Linear Congruential Method (LCM) - Lehmer 1951

Producing random numbers X_1, X_2, X_3 ... between 0 and m-1 with recursive operation:

$$X_{i+1}=(a*X_i+c) \bmod m$$

X_0: is the seed

a: multiplier

c: increment (c=0 multiplicative congruential method, otherwise mixed congruential method)

m: modulus

Lehmer, D. H. (1949). "Mathematical methods in large-scale computing units". *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*: 141–146. (journal version: *Annals of the Computation Laboratory of Harvard University*, Vol. 26 (1951)).

Linear Congruential Method (LCM) – Lehmer 1951

LCM generates random integer numbers between 0 and m-1

Random numbers R_1, R_2, R_3 , ... between 0 and 1 can be generated as follows:

$$R_i = X_i / m, \; i = 1,2,3,...$$

The properties of all R depends heavily on the selection of X_0,a,c,m.

Linear Congruential Method (LCM) - Example

Example: X_0=27,a=17,c=43,m=100

X 0 = 27

X 1 = (17 * 27 + 43) mod 100 = 2 -> R 1 = 0.02

X 2 = (17 * 2 + 43) mod 100 = 77 -> R 2 = 0.77

X 3 = (17 * 77 + 43) mod 100 = 52 -> R 3 = 0.52

X 4 = (17 * 52 + 43) mod 100 = 27 -> R 4 = 0.27

X 5 = (17 * 27 + 43) mod 100 = 2 -> R 5 = 0.02

X 6 = (17 * 2 + 43) mod 100 = 77 -> R 6 = 0.77

X 7 = (17 * 77 + 43) mod 100 = 52 -> R 7 = 0.52

X 8 = (17 * 52 + 43) mod 100 = 27 -> R 8 = 0.27

X 9 = (17 * 27 + 43) mod 100 = 2 -> R 9 = 0.02

X 10 = (17 * 2 + 43) mod 100 = 77 -> R 10 = 0.77

X 11 = (17 * 77 + 43) mod 100 = 52 -> R 11 = 0.52

X 12 = (17 * 52 + 43) mod 100 = 27 -> R 12 = 0.27

The average element of the list is: 0.3938095238095238

Linear Congruential Method (LCM) – Code Example Advanced

```
a = 17
c = 43
m = 100
seed = 27

def avg(locallist):
    sum = 0
    for element in locallist:
        sum += element
    print ('The average element of the list is: ' +
str((sum/(len(locallist)*1.0))/m))

def seedLCG(initVal):
    global rand
    rand = initVal

def lcg():
    global rand
    rand = (a*rand + c) % m
    return rand
```

```
liste = []

for i in range(m+5):
    if i==0:
        seedLCG(seed)
        liste.append(seed)
        print("X",i," = ",seed)

    if i>0:
        random_number=lcg()
        liste.append(random_number)
        if i < 15 or i >= m:
            print("X",i," = (",a,"*",liste[i-1],"+",c,") mod ",m," =
",liste[i],'-> R',i," = ",liste[i]/m)

print(avg(liste))
```

Linear Congruential Method (LCM) – Code Example Simple

```
#######Initialization#######
x0 = 27
a = 17
c = 43
m = 100

#######Adding Seed-Value#####
randlist = []
randlist.append(x0)

#######Generate and fill with random numbers#######
for count in range(m+5):
    randlist.append((a*randlist[-1]+c)%m)
print(randlist)
```

Linear Congruential Method (LCM)

Values produced are discrete and not continuous, but for large m this does not have a huge impact.

Besides uniformity and independence two further aspects have to be considered:

- Maximum density
- Maximum period

Linear Congruential Method (LCM) - Advanced

In [Law 2007] it has been shown that the maximal period can be achieved by the proper choice of X_0,a,c,m:

- Period equals m, if m is power of 2, e.g. $m=2^b$, and c!=0, if c is relatively prime to m (the greatest common factor of c and m is 1) and a=1+4k (k any integer); a is referred to full-period-multiplier if period is m.

- Let‘s try
  - $m=2^b=2^4=16$
  - c=43
  - a=1+4k=1+4*4=17

Linear Congruential Method (LCM) - Advanced

Example: X_0=27,a=17,c=43,m=16

X 0 = 27
X 1 = (17 * 27 + 43) mod 16 = 6 -> R 1 = 0.375
X 2 = (17 * 6 + 43) mod 16 = 1 -> R 2 = 0.0625
X 3 = (17 * 1 + 43) mod 16 = 12 -> R 3 = 0.75
X 4 = (17 * 12 + 43) mod 16 = 7 -> R 4 = 0.4375
X 16 = (17 * 0 + 43) mod 16 = 11 -> R 16 = 0.6875
X 17 = (17 * 11 + 43) mod 16 = 6 -> R 17 = 0.375
X 18 = (17 * 6 + 43) mod 16 = 1 -> R 18 = 0.0625
X 19 = (17 * 1 + 43) mod 16 = 12 -> R 19 = 0.75
X 20 = (17 * 12 + 43) mod 16 = 7 -> R 20 = 0.4375

The average element of the list is: 0.5148809523809523

Linear Congruential Method (LCM) - Advanced

In [Law 2007] it has been shown that the maximal period can be achieved by the proper choice of X_0,a,c,m:

- Period equals $m/4=2^{b-2}$, if m is power of 2, e.g. $m=2^b$, and c=0, if a=3+8k (or a=5+8k with k any integer) and X0 is odd
- Let‘s try
  - $m=2^b=2^5=32$
  - c=0
  - a=3+8k=3+8*4=35
  - $X_0=27$

Linear Congruential Method (LCM) - Advanced

Example: X_0=27,a=35,c=0,m=32

X 0 = 27
X 1 = (35 * 27 + 0) mod 32 = 17 -> R 1 = 0.53125
X 2 = (35 * 17 + 0) mod 32 = 19 -> R 2 = 0.59375
X 3 = (35 * 19 + 0) mod 32 = 25 -> R 3 = 0.78125
X 4 = (35 * 25 + 0) mod 32 = 11 -> R 4 = 0.34375
X 5 = (35 * 11 + 0) mod 32 = 1 -> R 5 = 0.03125
X 6 = (35 * 1 + 0) mod 32 = 3 -> R 6 = 0.09375
X 7 = (35 * 3 + 0) mod 32 = 9 -> R 7 = 0.28125
X 8 = (35 * 9 + 0) mod 32 = 27 -> R 8 = 0.84375
X 9 = (35 * 27 + 0) mod 32 = 17 -> R 9 = 0.53125
X 10 = (35 * 17 + 0) mod 32 = 19 -> R 10 = 0.59375
X 11 = (35 * 19 + 0) mod 32 = 25 -> R 11 = 0.78125
X 12 = (35 * 25 + 0) mod 32 = 11 -> R 12 = 0.34375
The average element of the list is: 0.4375

Linear Congruential Method (LCM) - Advanced

Example: $\mathbf{X_0=26}$,a=35,c=0,m=32 (JUST PROOF X0 HAS TO BE ODD!)

X 0 = 26
X 1 = (35 * 26 + 0) mod 32 = 14 -> R 1 = 0.4375
X 2 = (35 * 14 + 0) mod 32 = 10 -> R 2 = 0.3125
X 3 = (35 * 10 + 0) mod 32 = 30 -> R 3 = 0.9375
X 4 = (35 * 30 + 0) mod 32 = 26 -> R 4 = 0.8125
X 5 = (35 * 26 + 0) mod 32 = 14 -> R 5 = 0.4375
X 6 = (35 * 14 + 0) mod 32 = 10 -> R 6 = 0.3125
X 7 = (35 * 10 + 0) mod 32 = 30 -> R 7 = 0.9375
X 8 = (35 * 30 + 0) mod 32 = 26 -> R 8 = 0.8125
The average element of the list is: 0.625

Linear Congruential Method (LCM) - Advanced

In [Law 2007] it has been shown that the maximal period can be achieved by the proper choice of X_0, a, c, m:

- Period equals m-1, if m is prime and c=0, if a has the property that the smallest integer k such that a^k-1 is divisible by m is k=m-1h
- Let‘s try
    - $m=2^b-1=2^5-1=31$
    - c=0
    - a=17
    - $X_0=26$

Linear Congruential Method (LCM) - Advanced

Example: X_0=26,a=17,c=0,m=31

X 0 = 26
X 1 = (17 * 26 + 0) mod 31 = 8 -> R 1 = 0.25806451612903225
X 2 = (17 * 8 + 0) mod 31 = 12 -> R 2 = 0.3870967741935484
X 3 = (17 * 12 + 0) mod 31 = 18 -> R 3 = 0.5806451612903226
X 4 = (17 * 18 + 0) mod 31 = 27 -> R 4 = 0.8709677419354839
X 5 = (17 * 27 + 0) mod 31 = 25 -> R 5 = 0.8064516129032258
X 6 = (17 * 25 + 0) mod 31 = 22 -> R 6 = 0.7096774193548387
X 29 = (17 * 15 + 0) mod 31 = 7 -> R 29 = 0.22580645161290322
X 30 = (17 * 7 + 0) mod 31 = 26 -> R 30 = 0.8387096774193549
X 31 = (17 * 26 + 0) mod 31 = 8 -> R 31 = 0.25806451612903225
X 32 = (17 * 8 + 0) mod 31 = 12 -> R 32 = 0.3870967741935484
X 33 = (17 * 12 + 0) mod 31 = 18 -> R 33 = 0.5806451612903226
X 34 = (17 * 18 + 0) mod 31 = 27 -> R 34 = 0.8709677419354839
The average element of the list is: 0.5206093189964158

Linear Congruential Method (LCM) - Advanced

- 2^{31}-1 is a Mersenne Prime number: a prime number which is one less than an integer power of 2
- The lowest multiplier for an LCG with a modulus constant of (2^{31} - 1) is 16807 (based on research in 1969[1])
- 47271 or 69621 might be better options (based on research conducted in 1988 [2])

[1] WH Payne, JR Rabung, TP Bogyo, "Coding the Lehmer pseudo-random number generator", Communications of the ACM, February 1969, Vol 12 Number 2 85-86

[2] Stephen K. Park and Keith W. Miller "Random Number Generators: Good Ones are Hard to Find", Communications of the ACM, Oct 1988, Vol 31 Number 10 1192-1201

Combined Linear Congruential Generators

- Even random number generators with large periods such as 2^{31}-1 are not sufficient for certain application such as cryptographic applications
- Combined Linear Congruential generators are able to combine multiple generators with larger periods
- L‘Ecuyer (1988)
- Example let‘s combine two generators
  - a_1=40014, m_1=2147483563
  - a_2=40692, m_2=2147483399
- Select two seeds
  - $X_{1,0}$ in range[1:2147483562]
  - $X_{2,0}$ in range[1:2147483398]

Combined Linear Congruential Generators

Loop (j=0,j++)

$X_{1,j+1}=40014*X_{1,j} \bmod 2147483563$
$X_{2,j+1}=40692*X_{2,j} \bmod 2147483399$

$X_{j+1}=(X_{1,j+1}-X_{2,j+1}) \bmod 2147483562$ *// max(m1,m2) - 1*

$R_{j+1}=\frac{X_{j+1}}{2147483563}$ if $X_{j+1} > 0$

$R_{j+1}=\frac{2147483562}{2147483563}$ if $X_{j+1} = 0$

Combined Linear Congruential Generators

The combined generator has a new period of $(m_1-1)(m_2-1)$~$2x10^{18}$

EXERCISE: Extend the Python script for a single to a combined linear congruential generator.

Combined Linear Congruential Generators

```
#######Initialization#######
x10 = 16777
x20 = 97333
m1 = 2147483563
m2 = 2147483399
m = max(m1,m2) - 1
a1 = 40014
a2 = 40692
c1 = c2 = 43

generationCount = 100
#######Adding Seed-Value#####
randlist1 = []
randlist2 = []
randlistnew = []
randlist1.append(x10)
randlist2.append(x20)

#######Generate and fill with random numbers#######
for count in range(generationCount):
    randlist1.append((a1*randlist1[-1]+c1)%m1)
    randlist2.append((a2*randlist2[-1]+c2)%m2)
    randlistnew.append((randlist1[-1]-randlist2[-1])%m)

print(randlistnew)
```

Tausworthe Generator

- In order to generate random numbers for cryptographic use cases long random numbers are needed
- In contrast to be aforementioned examples here we produce a stream on bits (0,1) not numbers
- Later bits can be mapped into numbers
- Tuaswothe proposed the following structure in 1965

$$b_n = c_{q-1} b_{n-1} \times c_{q-2} b_{n-2} \times \ldots \times c_0 b_{n-q}$$

c and b are binary and x is the modulo 2 operator (EXCLUSIVE OR).

Tausworthe Generator

The Tausworthy generator will generator random numbers with a period of 2^q-1

Using the characteristic polynom:

$$X^q+c_{q-1}x^{q-1}+c_{q-2}x^{q-2}+\ldots+c_0$$

The period is given by the order of the *primitive polynom*

$$X^7+x^3+1$$

$$D^7b(n)+D^3b(n)+b(n)=0 \bmod 2$$

$$b_{n+7}+b_{n+3}+b_n=0 \text{ with } n=0,1,2,3,\ldots$$

$$b_{n+7}xb_{n+3}xb_n=0 \text{ with } n=0,1,2,3,\ldots$$

$$b_{n+7}=b_{n+3}xb_n \quad \text{with } n=0,1,2,3,\ldots$$

$$b_n=b_{n-4}xb_{n-7} \quad \text{with } n=7,8,9,10$$

Tausworthe Generator

Starting state:

$b_0=b_1=b_2=b_3=b_4=b_5=b_6=1$

Calculation:

$b_7=b_3xb_0=1x1=0$

$b_8=b_4xb_1=1x1=0$

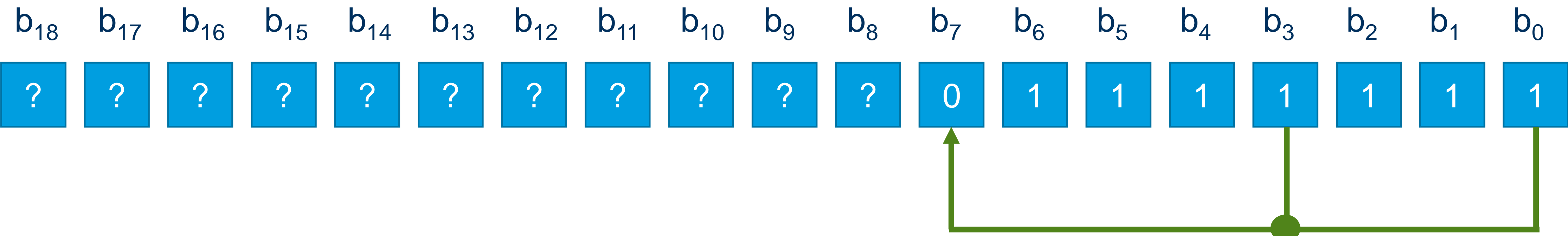
$b_9=b_5xb_2=1x1=0$

$b_{10}=b_6xb_3=1x1=0$

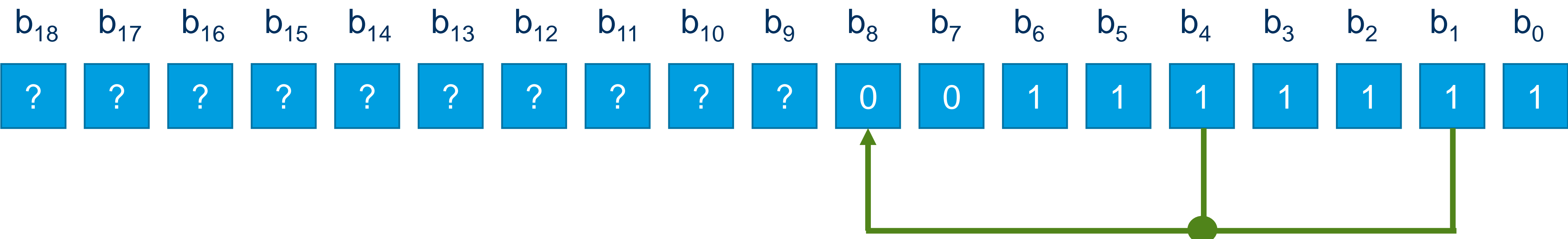
$b_{11}=b_7xb_4=0x1=1$

$b_{12}=b_8xb_5=0x1=1$

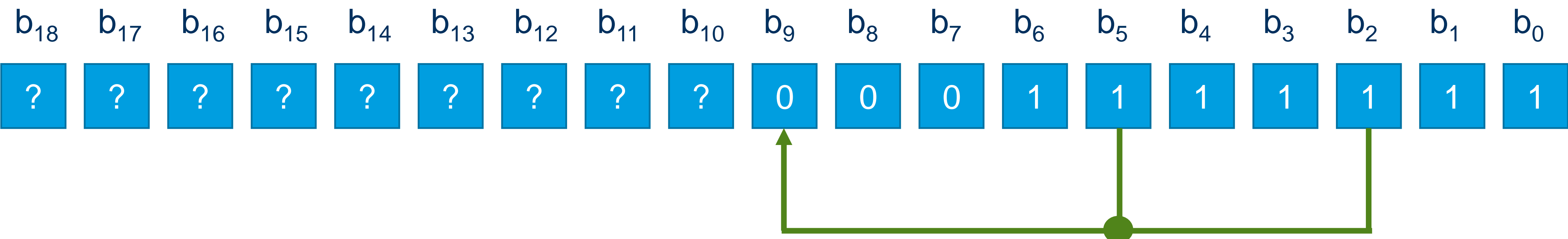
Tausworthe Generator



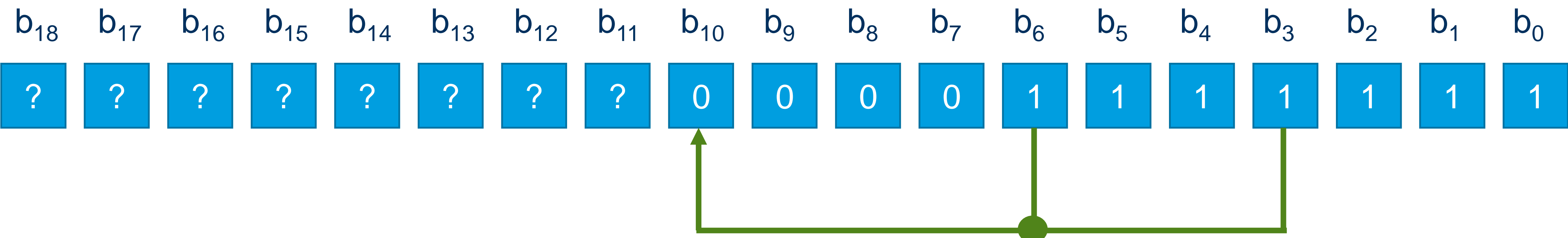
Tausworthe Generator



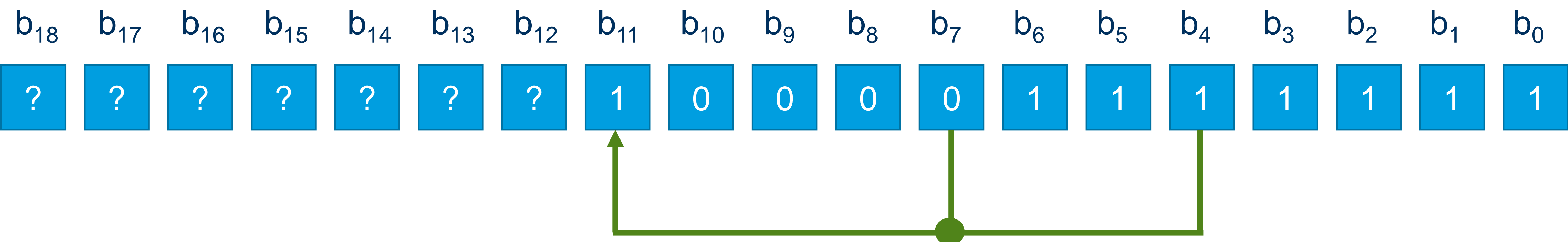
Tausworthe Generator



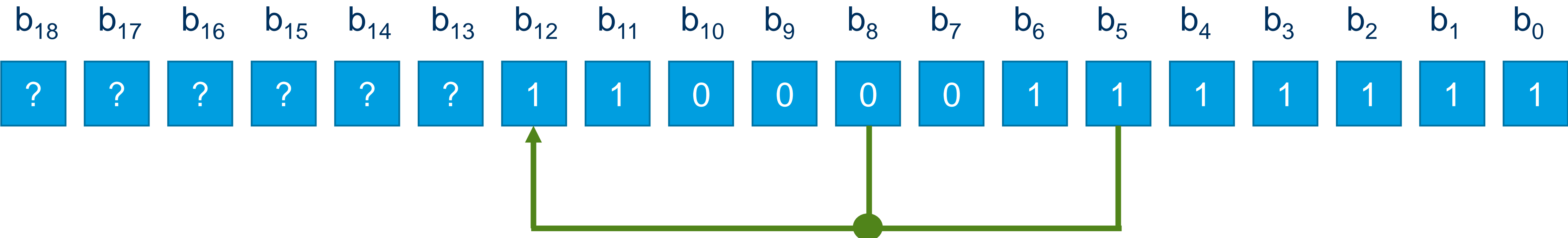
Tausworthe Generator



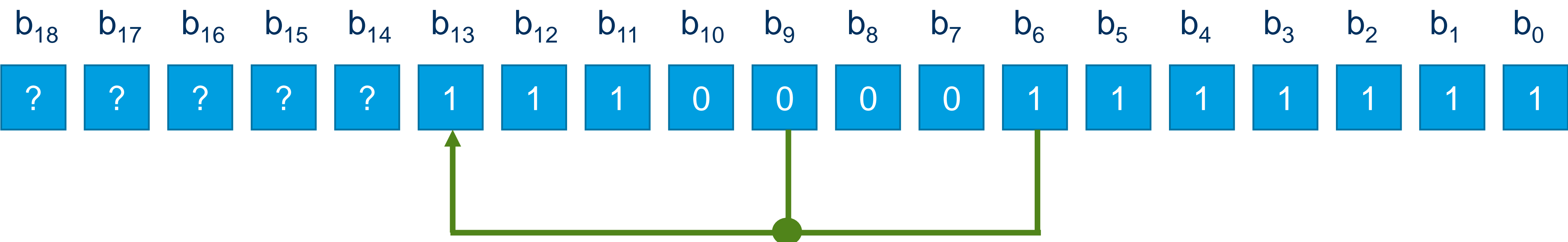
Tausworthe Generator



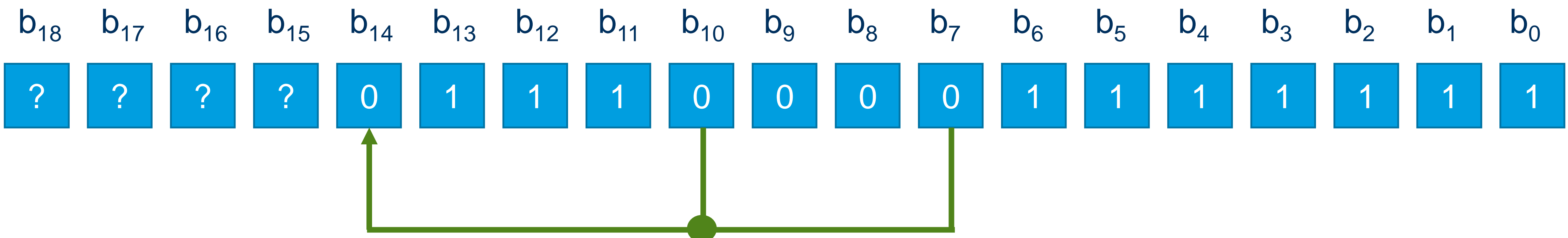
Tausworthe Generator



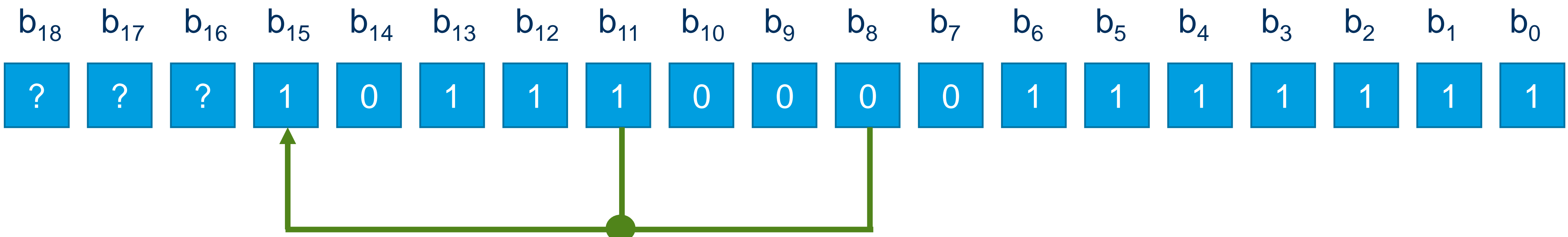
Tausworthe Generator



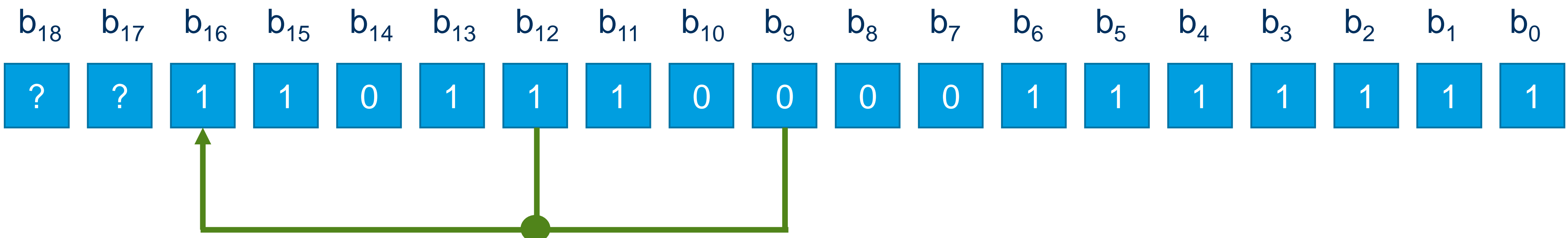
Tausworthe Generator



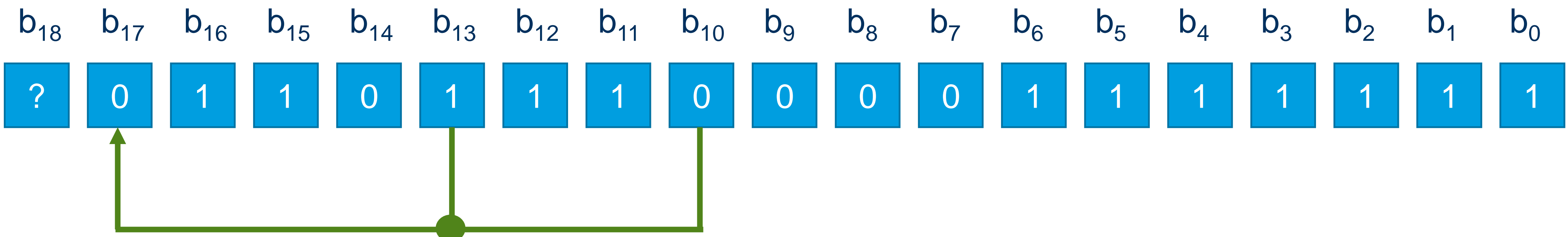
Tausworthe Generator



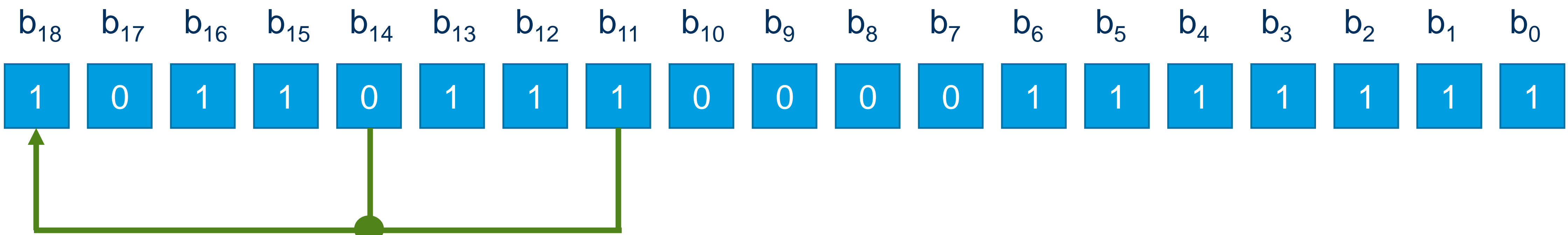
Tausworthe Generator



Tausworthe Generator



Tausworthe Generator



Tausworthe Generator: Python

```
result = []

def lfsr(seed, taps):
    sr, xor = seed, 0
    for count in range(len(sr)):
        if sr[count]=="0":
            result.append(0)
        else:
            result.append(1)
    for count in range(150):
        for t in taps:
            xor += int(sr[t-1])
            print("b_",count+t,"=",sr[t-1])
        if xor%2 == 0.0:
            xor = 0
        else:
            xor = 1
        print("result",xor)
        result.append(xor)
        sr, xor = str(xor) + sr[:-1], 0
        print(sr)

lfsr('1111111', (7,4))
print(result)
```

Tausworthe Generator

Result for the first 150 bits

[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0]

Tausworthe Generator

Result for the first 150 bits

[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0]

Tausworthe Generator

Result for the first 150 bits

[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0]

127

x^7+x^3+1 is a primitive polynomial

Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator of feed it though a distribution (uniform)

Random Number Benchmarks for uint8

Benchmark name

std_rand
boost_taus88_dist
boost_mt19937_dist
boost_mt19937_64_dist
boost_mt11213b_dist
boost_minstd_rand_dist
boost_kreutzer1986_dist
boost_knuth_b_dist

time

0 20 40 60 80 100

MB/s

Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator of feed it though a distribution (uniform)

Random Number Benchmarks for uint16

time

Benchmark name

std_mt19937_dist
std_mt19937_64_dist
std_minstd_rand_dist
std_minstd_rand0_dist
boost_taus88_dist
boost_mt19937_dist
boost_mt19937_64_dist
boost_mt11213b_dist
boost_minstd_rand_dist
boost_kreutzer1986_dist
boost_knuth_b_dist

0 20 40 60 80 100 120 140 160 180

MB/s

Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator of feed it though a distribution (uniform)

Random Number Benchmarks for uint32

time

Benchmark name

std_mt19937_raw
std_mt19937_dist
std_mt19937_64_dist
std_minstd_rand_raw
std_minstd_rand_dist
std_minstd_rand0_raw
std_minstd_rand0_dist
boost_taus88_raw
boost_taus88_dist
boost_mt19937_raw
boost_mt19937_dist
boost_mt19937_64_dist
boost_mt11213b_raw
boost_mt11213b_dist
boost_minstd_rand_raw
boost_minstd_rand_dist
boost_kreutzer1986_raw
boost_kreutzer1986_dist
boost_knuth_b_raw
boost_knuth_b_dist

0 100 200 300 400 500 600 700 800 900

MB/s

Random Generators

- The results show how many MB/s of random data can you generate with each algorithm
- *uintX* is how many X bits you generate at a time
- The *dist* and *raw* is whether just use then raw random number generator of feed it though a distribution (uniform)

Random Number Benchmarks for uint64

time

std_mt19937_dist
std_mt19937_64_raw
std_mt19937_64_dist
std_minstd_rand_dist
std_minstd_rand0_dist
boost_taus88_dist
boost_mt19937_dist
boost_mt19937_64_raw
boost_mt19937_64_dist
boost_mt11213b_dist
boost_minstd_rand_dist
boost_kreutzer1986_dist
boost_knuth_b_dist

Benchmark name

0 100 200 300 400 500 600 700 800 900

MB/s

Combination of Inter and Intra Flow Network Coding

J. Krigslund, J. Hansen, M. Hundeboll, F.H.P. Fitzek, and T. Larsen, “Core: Cope with more in wireless meshed networks,” in IEEE VTC2013-Spring: Cooperative Communication, Distributed MIMO and Relaying, Dresden, Germany, June 2013.

Combination of Inter and Intra Flow Network Coding

Inter flow network coding
XOR type of coding
COPE
CATWOMAN
...

Intra flow network coding
RLNC type of coding
MORE
...

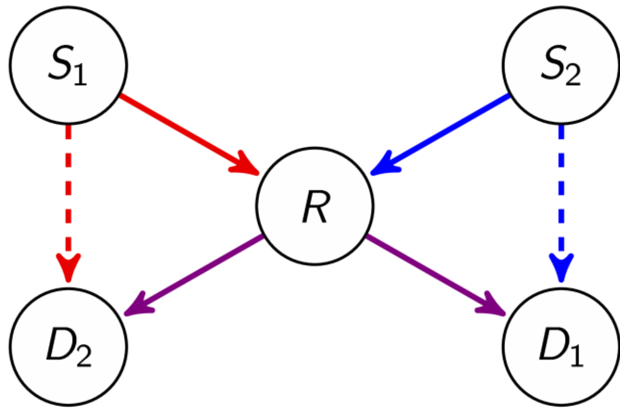
Inter & Intra flow network coding
Combine the best ...
CORE
...

Setup under investigation

Challenge: channel losses

Overhearing is critical for inter-session

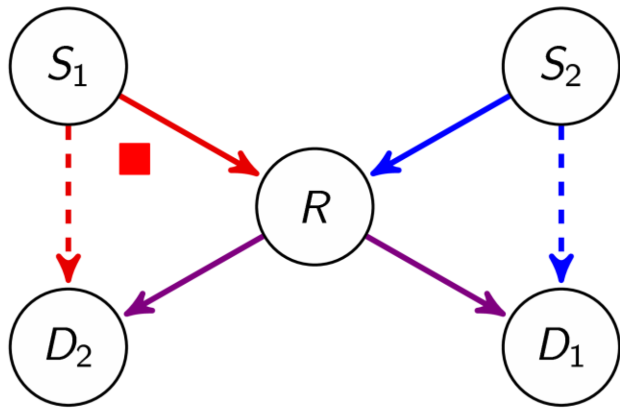
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

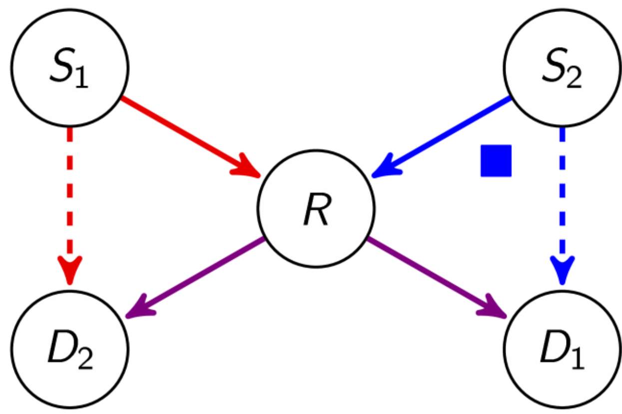
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

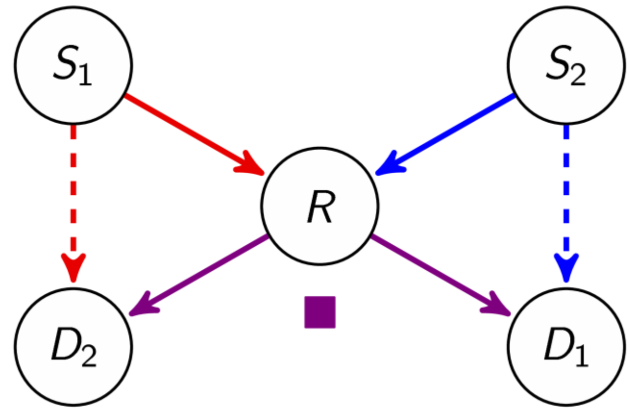
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

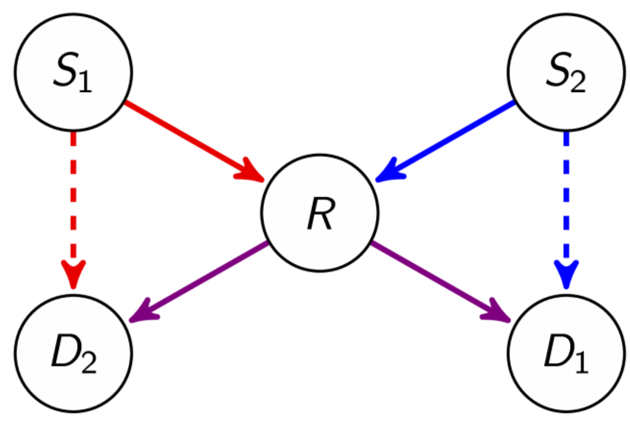
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

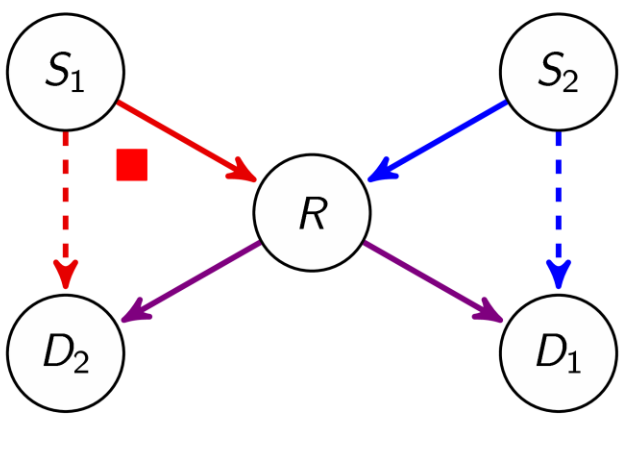
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

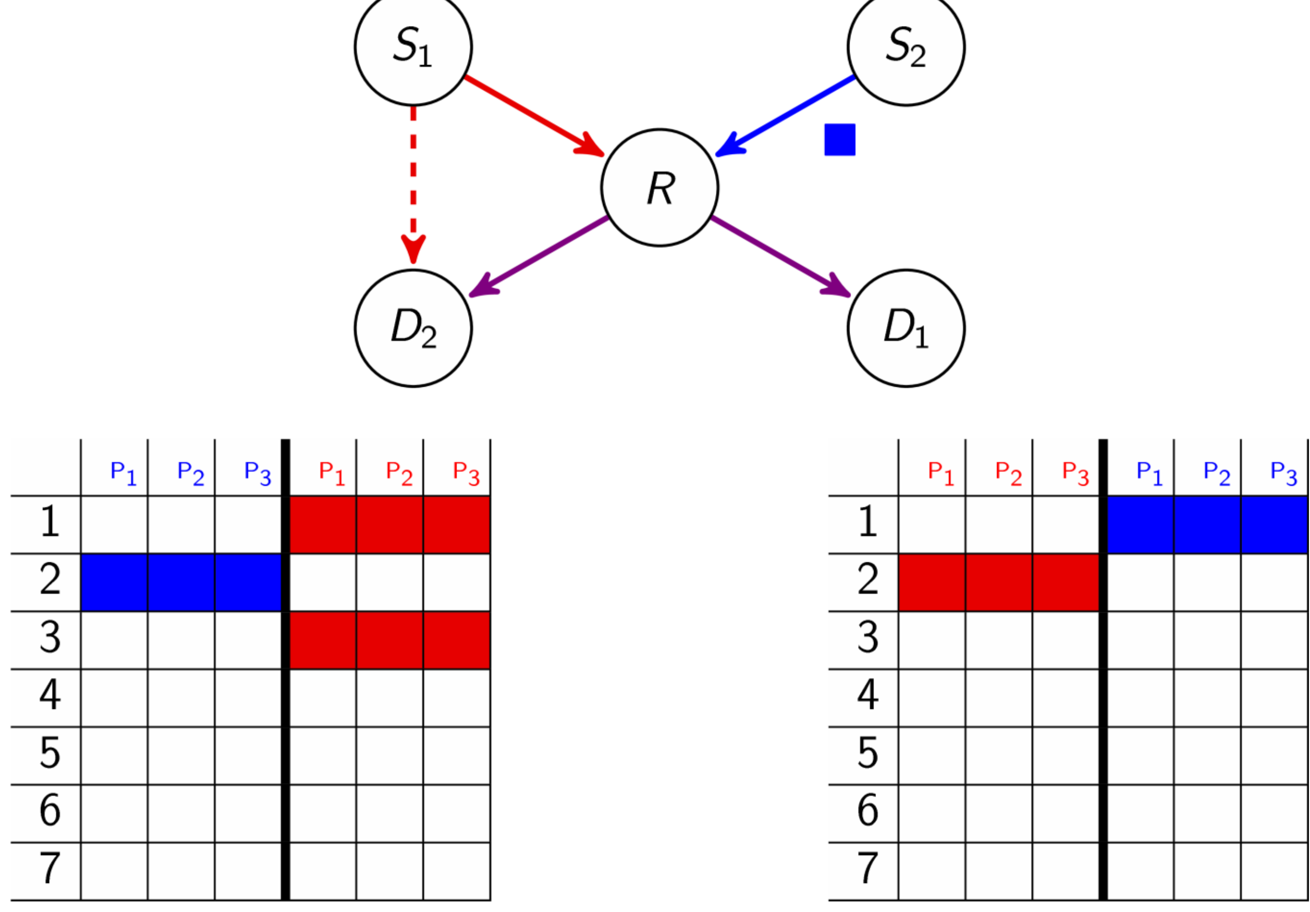
CORE



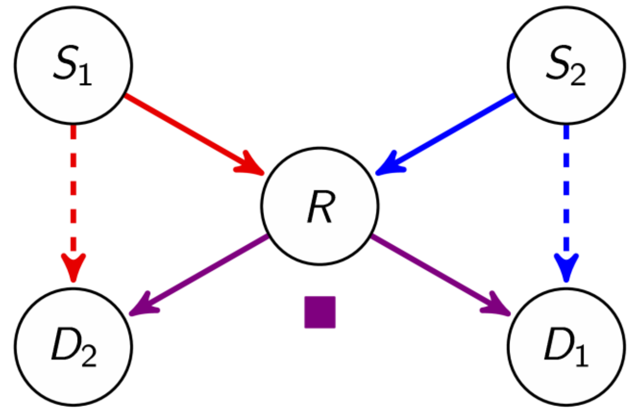
	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

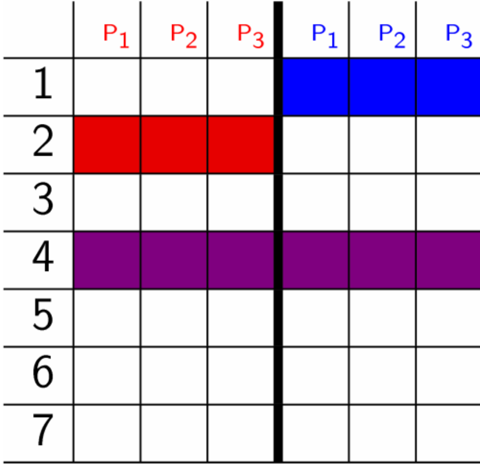
CORE



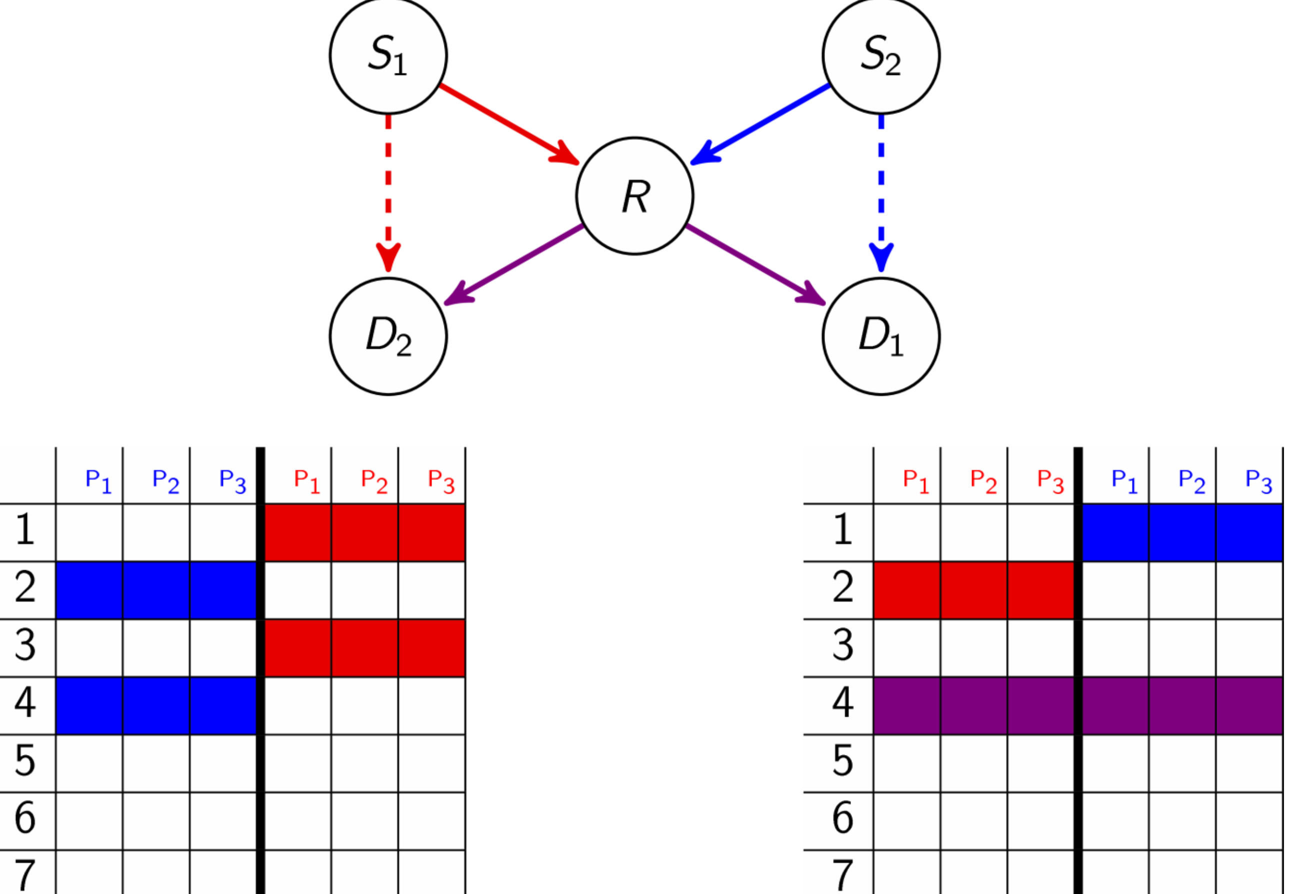
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						



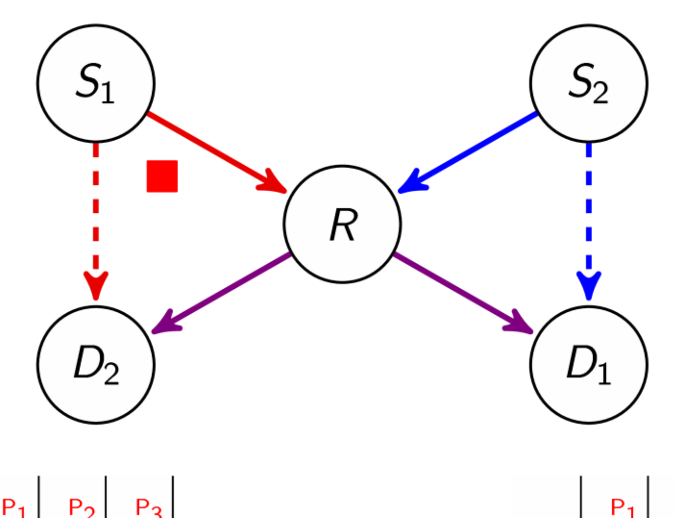
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

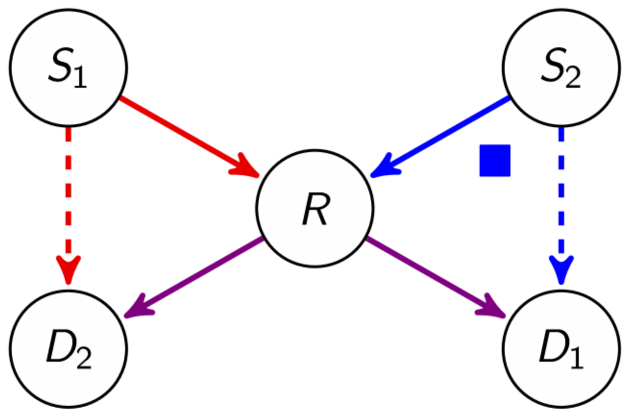
CORE



	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

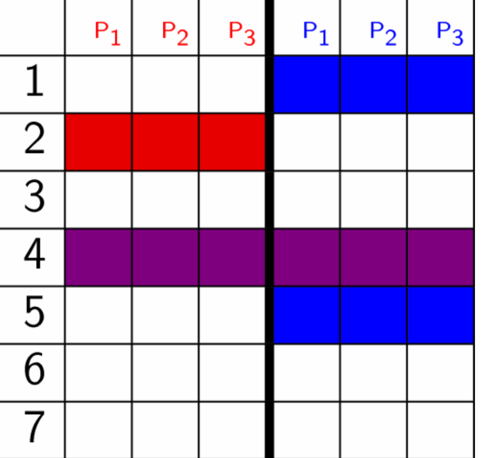
	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

CORE



Relay has full rank!
Stop sources …

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						



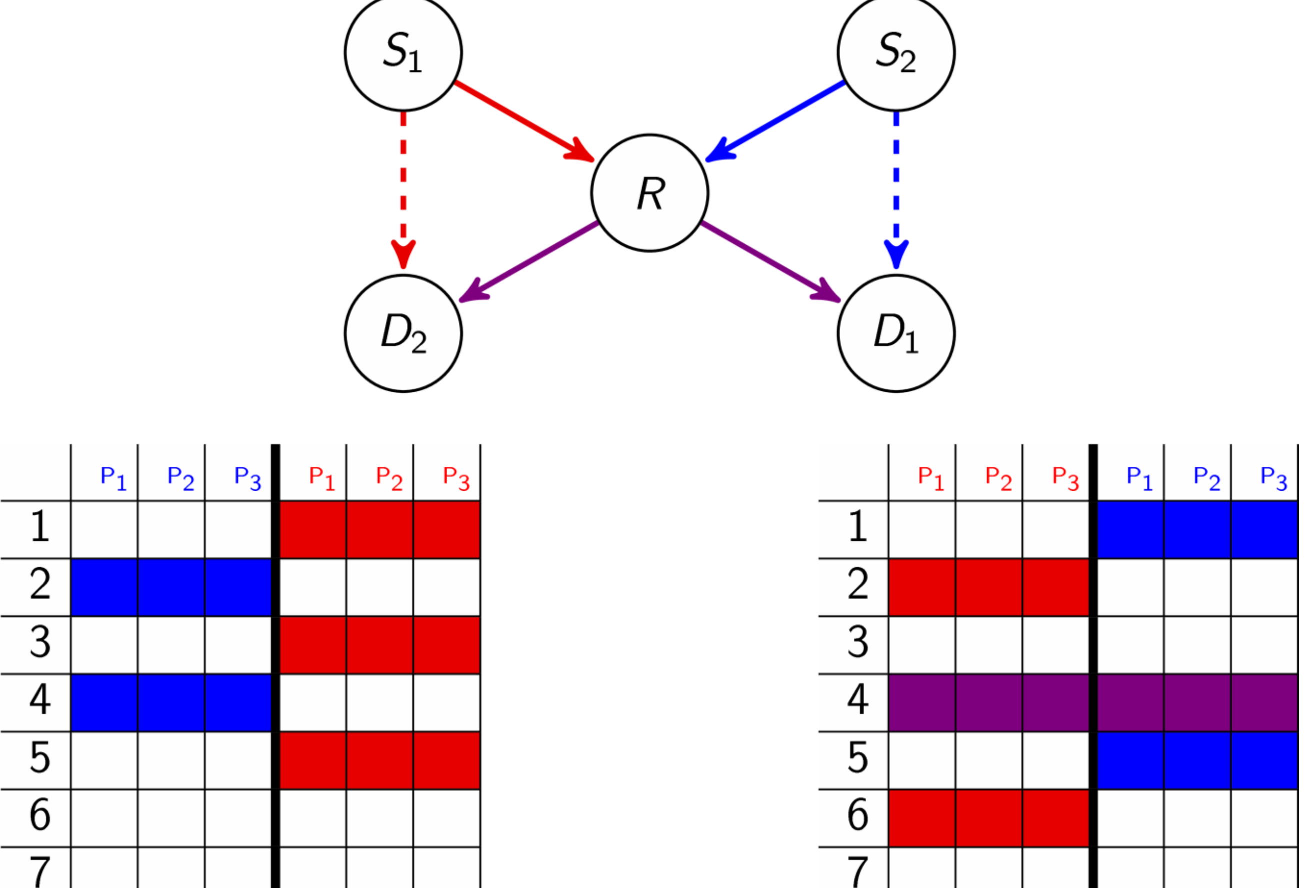
CORE

No retransmission

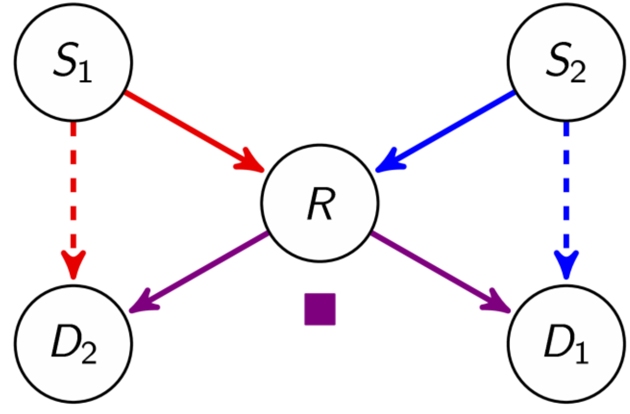
	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						

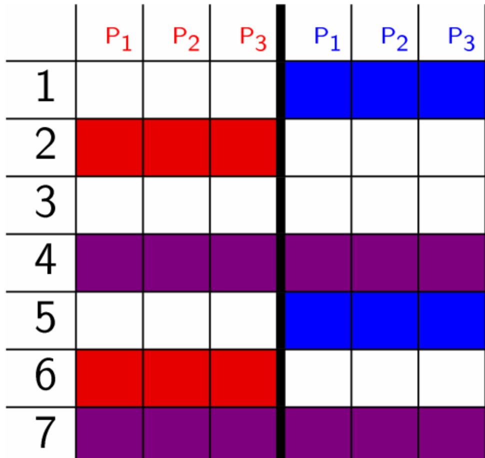
CORE



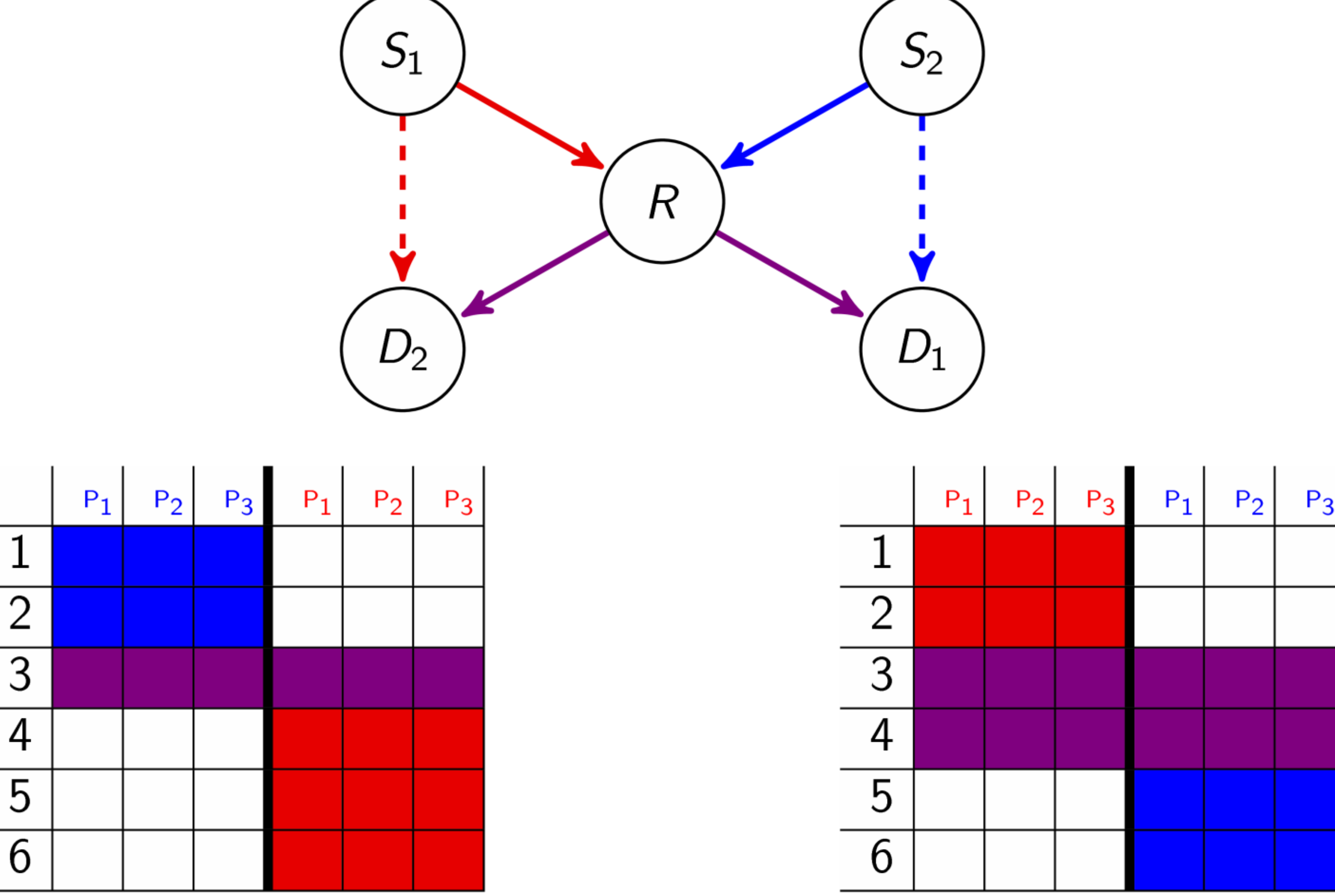
CORE



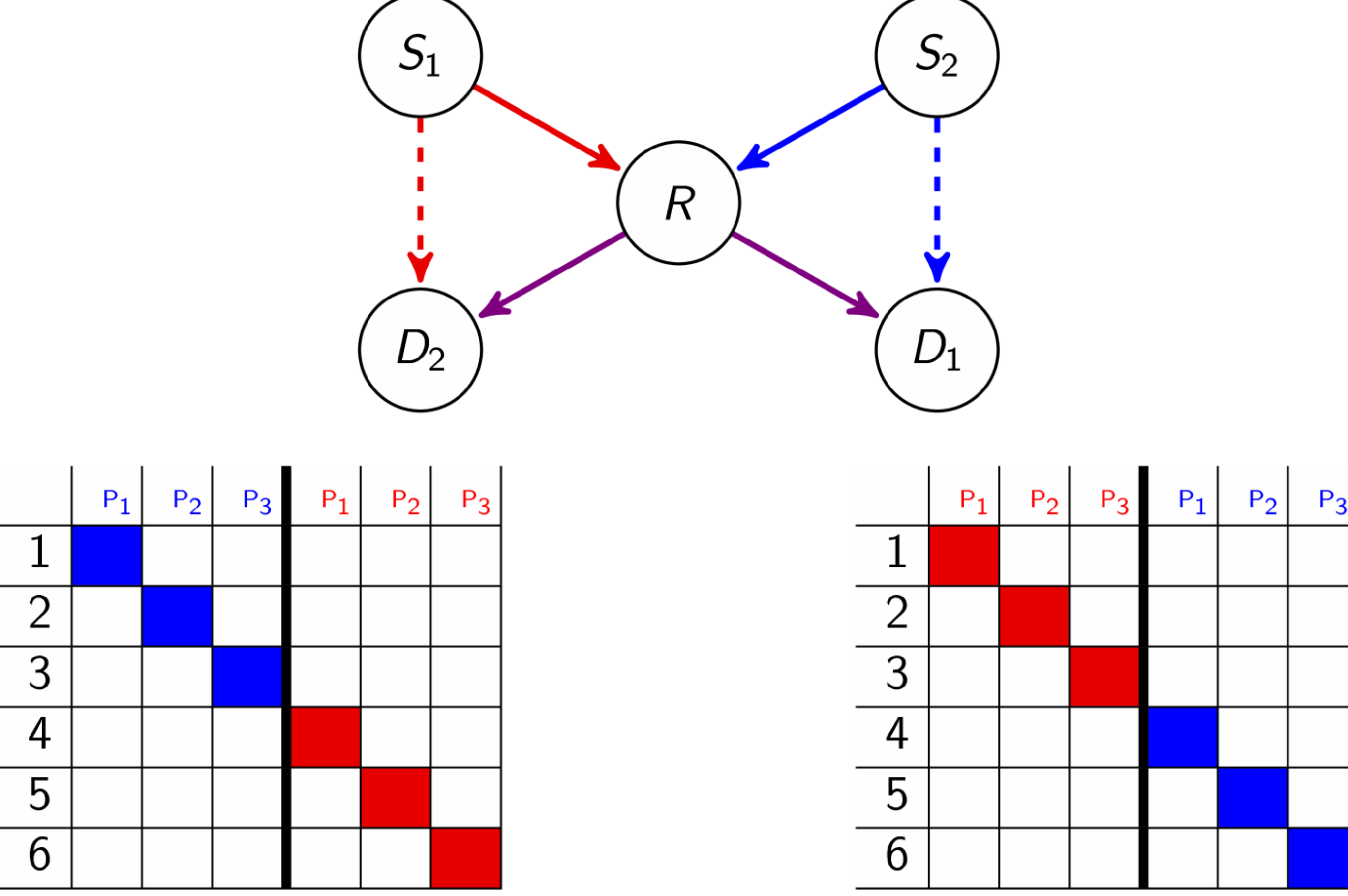
	P_1	P_2	P_3	P_1	P_2	P_3
1						
2						
3						
4						
5						
6						
7						



CORE



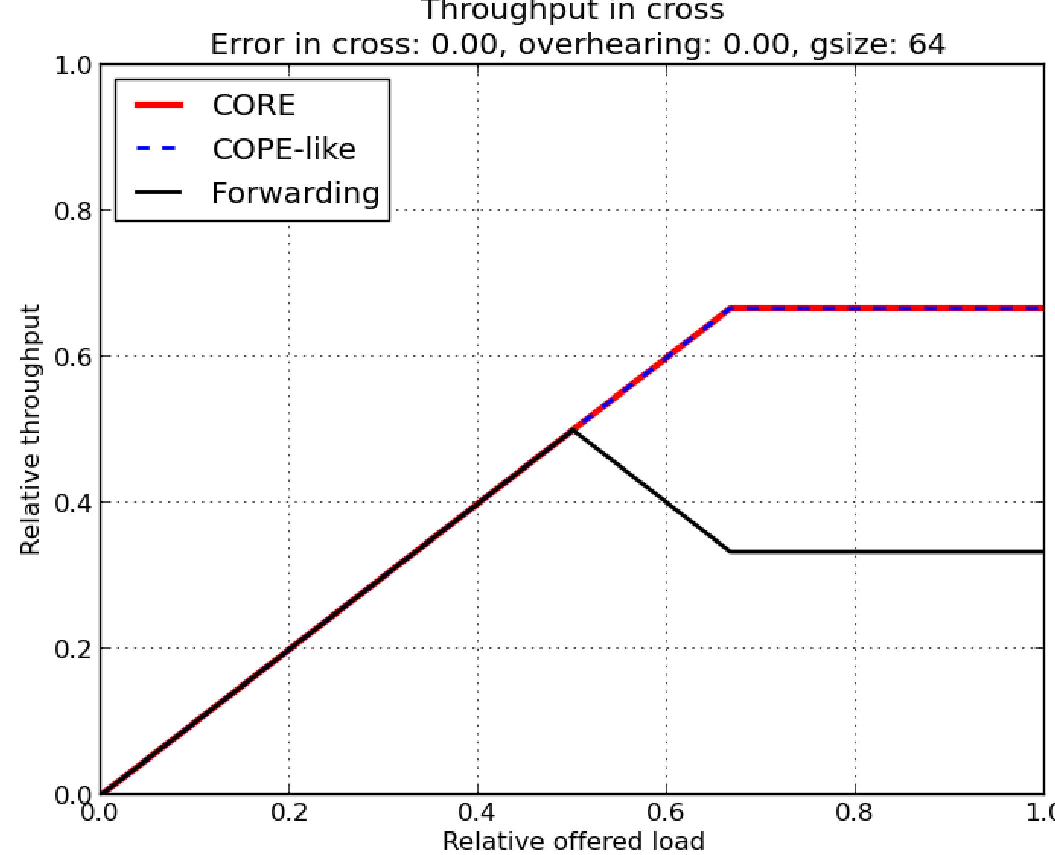
CORE



CORE Results

Throughput vs Offered Load

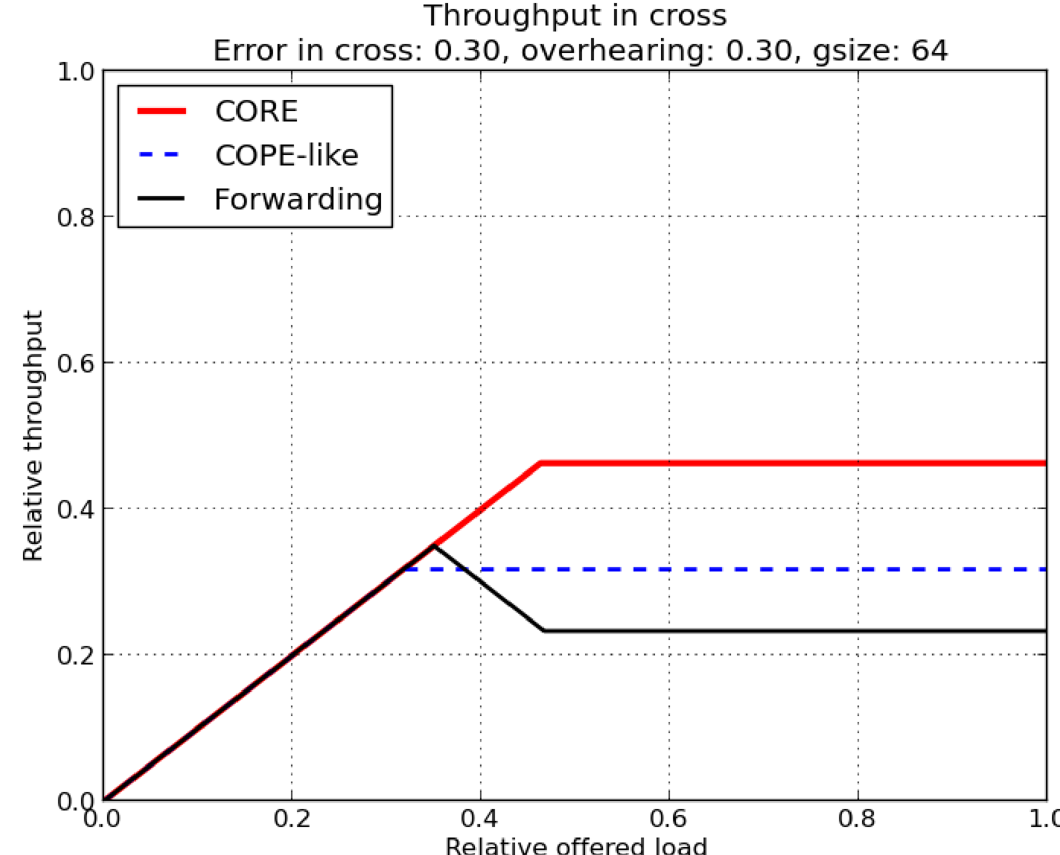
- Symmetric losses
- Loss probability = 0



CORE Results

Throughput vs Offered Load

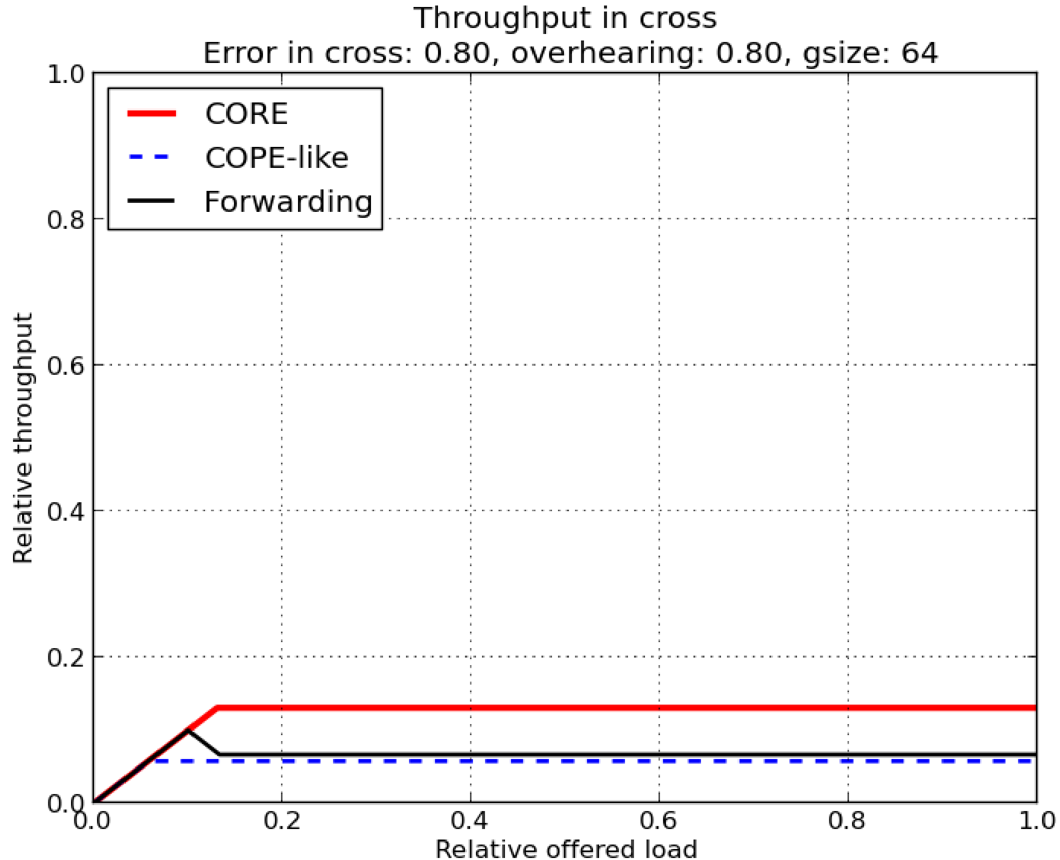
- Symmetric losses
- Loss probability = 0.3



CORE Results

Throughput vs Offered Load

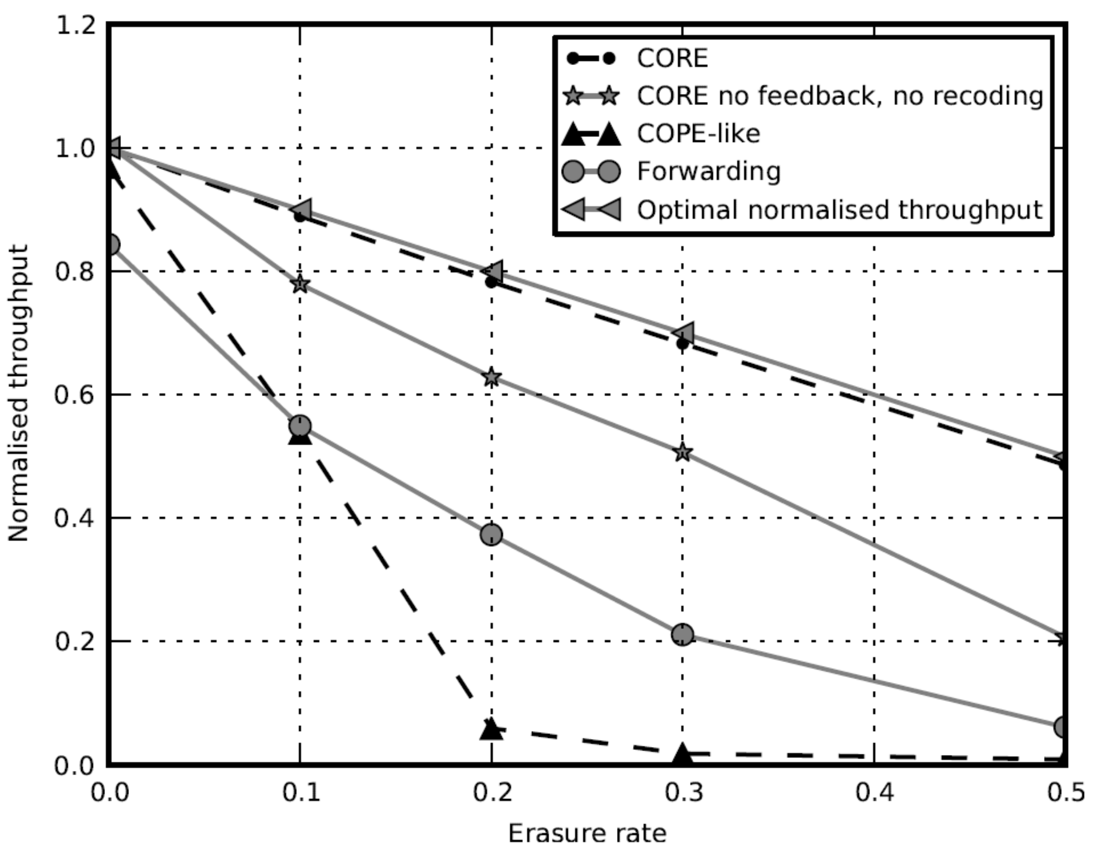
- Symmetric losses
- Loss probability = 0.8



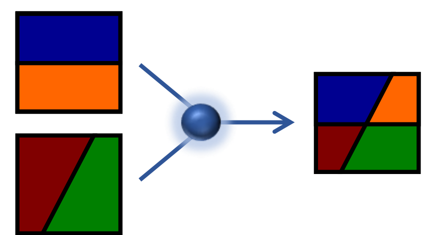
CORE Results

Throughput vs Losses

- Overall performance

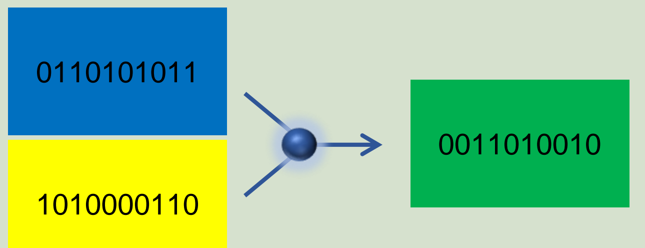


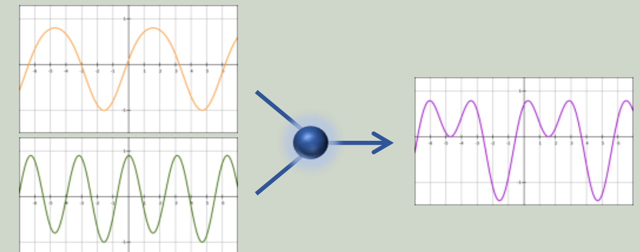
Different Forms of Network Coding



	Static Networks (planning possible)	Dynamic Networks (no planning possible)
Digital domain	*CRA I:* Digital Inter Session NC e.g.: Internet of Things (IoT) protocols, ...	*CRA III:* Digital Intra Session NC e.g.: distributed storage complexity, heterogeneity, delay, ...
Analog domain	*CRA II:* Analog Inter Session NC e.g.: wireless mesh networks multiple signals, real world, ...	*CRA IV:* Analog Intra Session NC e.g.: cooperative beamforming multiple signals, real world, ...

X-over





Analog Intra Flow

The missing link

Analog Intra Flow

- Main idea to use mobile device beamforming to create random superposition of signals in the analog domain and treat this as coded information.
- Inspired by former work on beamforming for COTS devices.

(12) **United States Patent**
Fitzek et al.

(10) **Patent No.:** **US 9,077,398 B2**
(45) **Date of Patent:** **Jul. 7, 2015**

(54) **NETWORK CODING BY BEAM FORMING**

(75) Inventors: **Frank Fitzek**, Aalborg (DK); **Janus Heide**, Aalborg (DK); **Morten Pedersen**, Aalborg (DK)

(73) Assignee: **NOKIA TECHNOLOGIES OY**, Espoo (FI)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **13/984,892**

(22) PCT Filed: **Feb. 22, 2011**

(86) PCT No.: **PCT/IB2011/000362**
§ 371 (c)(1), (2), (4) Date: **Aug. 12, 2013**

(87) PCT Pub. No.: **WO2012/114141**
PCT Pub. Date: **Aug. 30, 2012**

(65) **Prior Publication Data**
US 2013/0315338 A1 Nov. 28, 2013

(51) **Int. Cl.**

H04B 7/04	(2006.01)
H04B 7/06	(2006.01)
H04L 1/06	(2006.01)
H04B 7/02	(2006.01)
H04L 1/00	(2006.01)

(52) **U.S. Cl.**
CPC ***H04B 7/0408*** (2013.01); ***H04B 7/0617***

(56) **References Cited**

U.S. PATENT DOCUMENTS

2007/0149117 A1	6/2007	Hwang et al.
2007/0155336 A1	7/2007	Nam et al.

(Continued)

FOREIGN PATENT DOCUMENTS

WO	2009026695 A1	3/2009
WO	2010019340 A1	2/2010
WO	2011035797 A1	3/2011

OTHER PUBLICATIONS

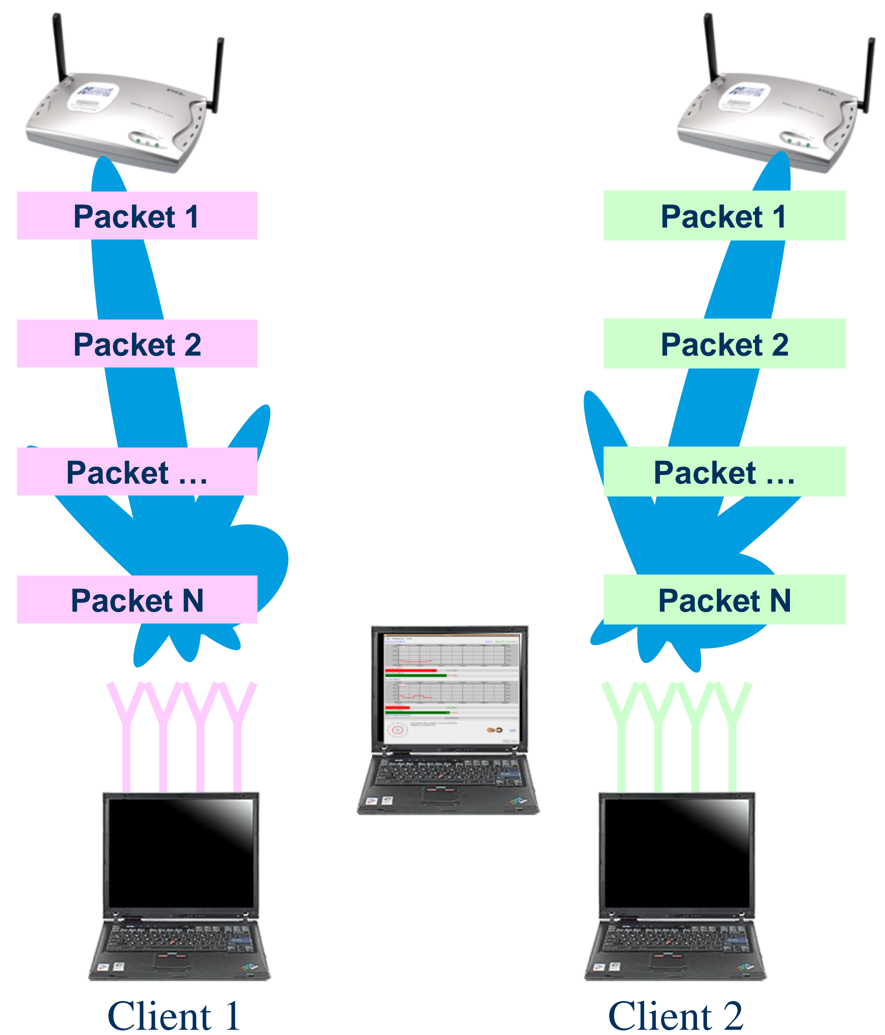
International Search Report and Written Opinion received for corresponding Patent Cooperation Treaty Application No. PCT/IB2011/000362, dated Dec. 1, 2011, 13 pages.

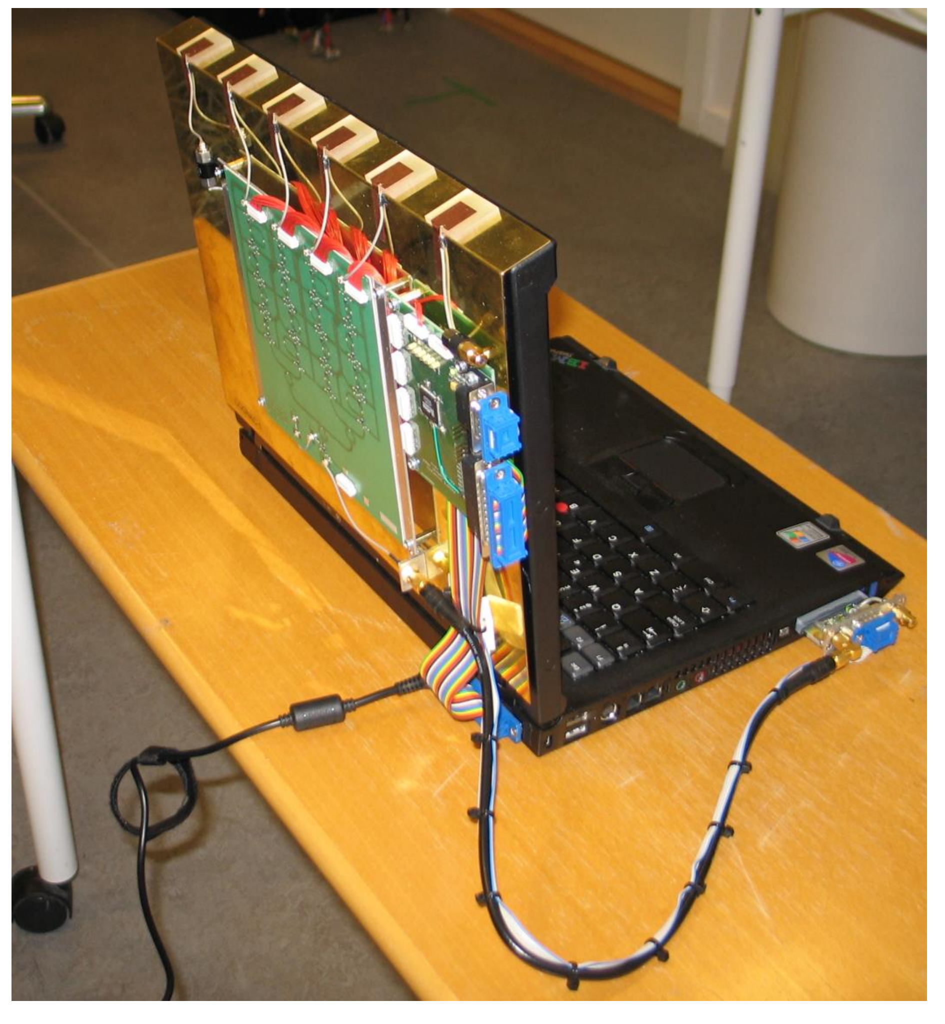
Primary Examiner — Leon Flores
(74) *Attorney, Agent, or Firm* — Squire Patton Boggs (US) LLP

(57) **ABSTRACT**

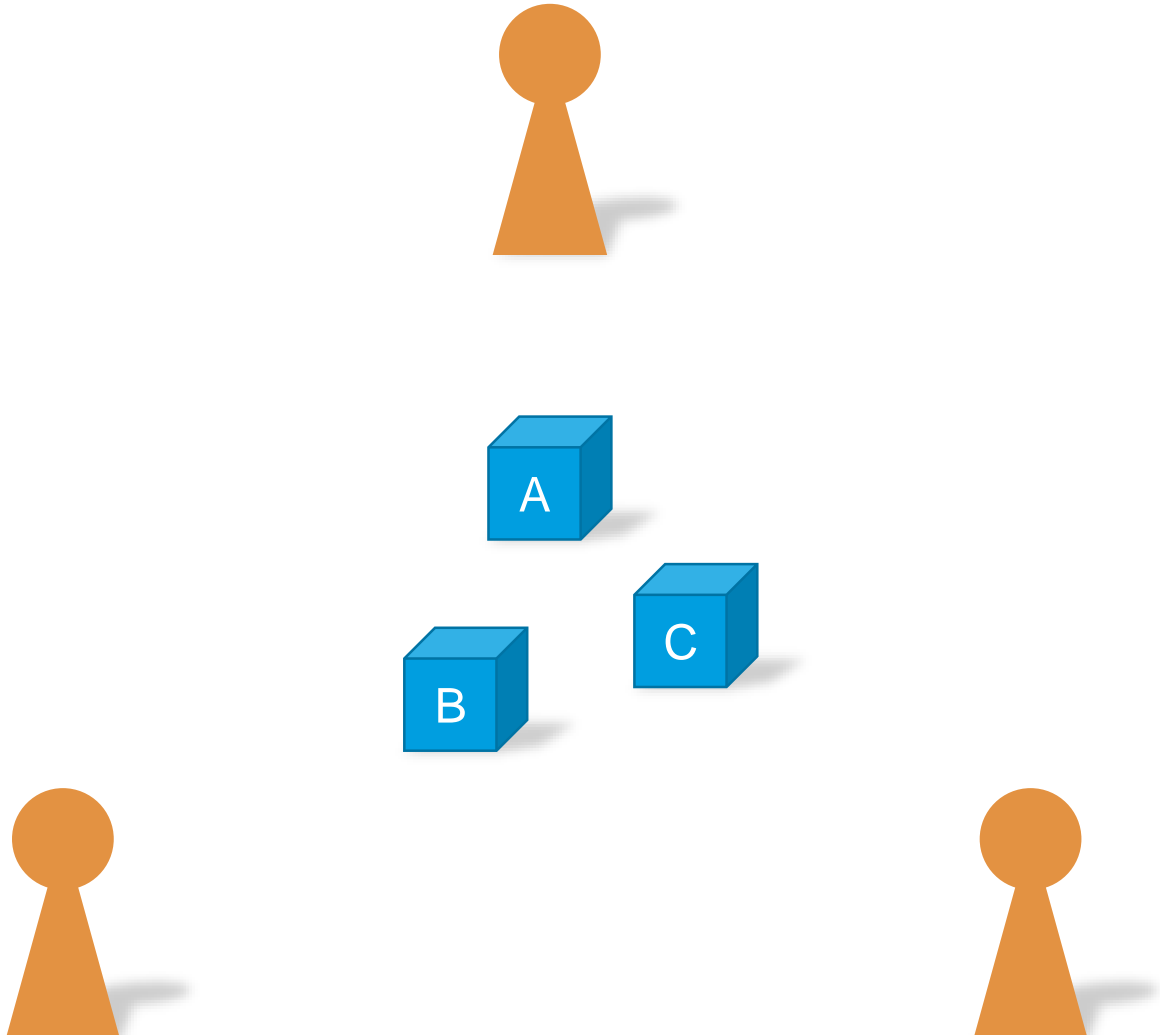
Network coding by beam forming in networks, for example, in single frequency networks, can provide aid in increasing spectral efficiency. When network coding by beam forming and user cooperation are combined, spectral efficiency gains may be achieved. According to certain embodiments, a method includes operating a user equipment of a plurality of user equipment in a network comprising a plurality of access points. The method also includes the user equipment forming a beam. The method further receives processing received signals from at least one of the plurality of access points at the user equipment. The forming the beam is configured to let different user equipment of the plurality of user equipment to

Cooperative Beamforming





Main Idea of Analog Network Coding with Beamforming



Finite Fields

Why finite Fields?

Linear Network Coding

Coded packets: linear combinations of original packets

Source

- Injects packets into the network (coded or uncoded)

Intermediate nodes

- Recombine with packets in their buffer

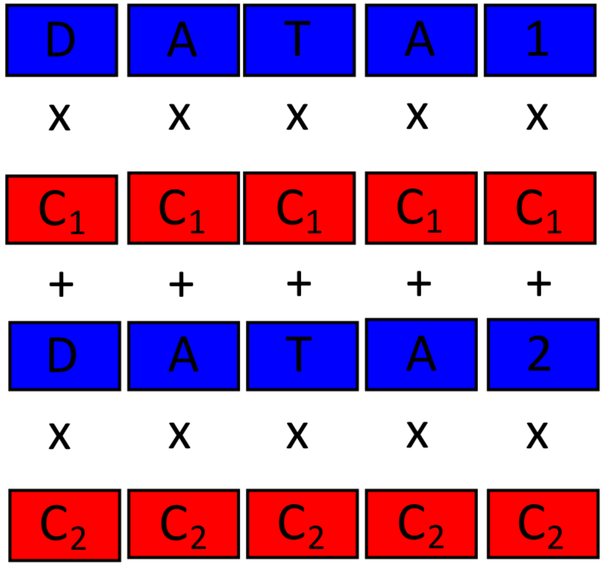
Receivers

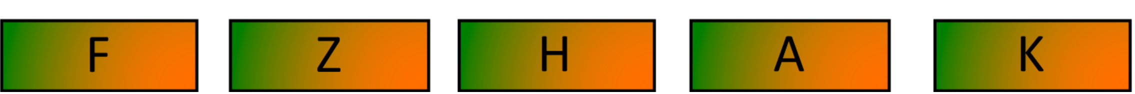
- Decode if enough linear comb. are received

What are finite fields? Why do we need them?

Finite Fields – do we need them?

Data representation: linear combination of multiple packets could result in large payloads





Goal:

- Linear combinations of packets should result in packets

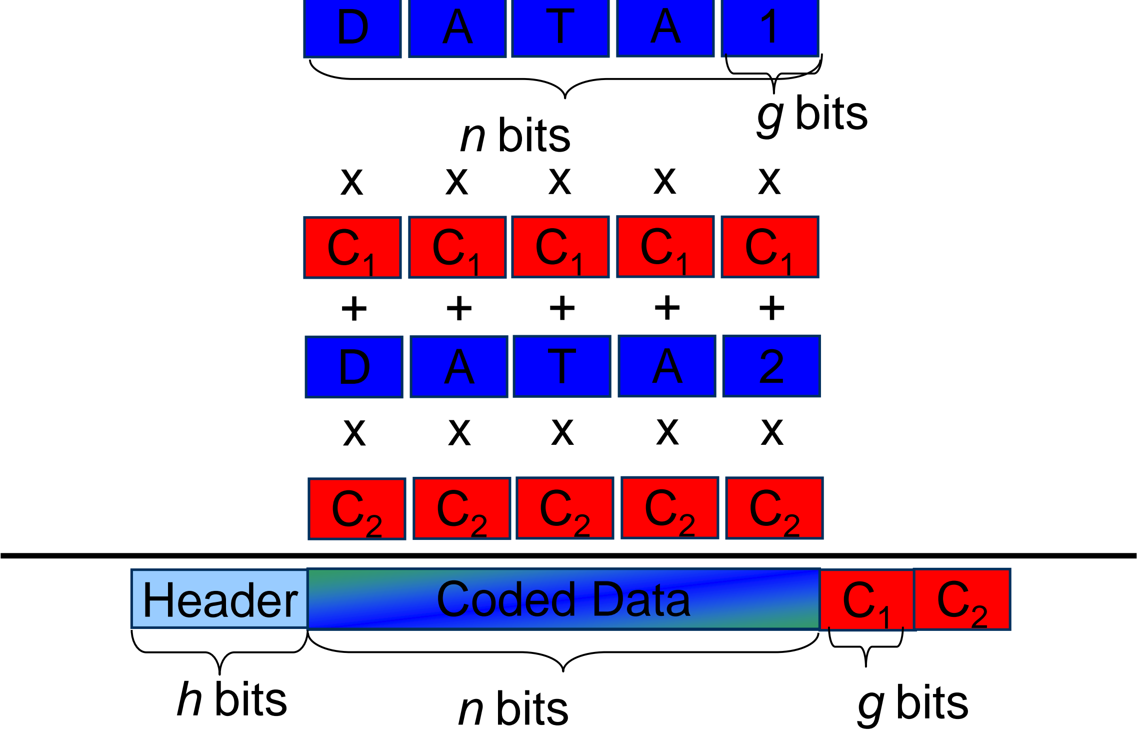
Generating a Coded Packet

- 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

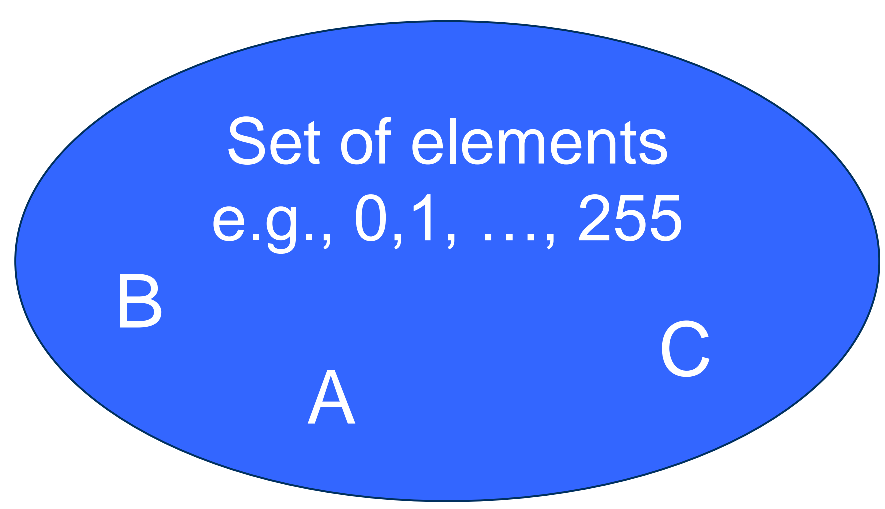


Finite Fields – What are they?

Operations:

Addition, Multiplication

Property: closure



+ , x

Groups, Rings, Fields

Field

Integral domain

Commutative ring

Ring

Abelian group

Group

Groups, Rings, Fields

Groups

- A group G, denoted {G, * } is a set of elements with a binary operation * that associates each ordered pair (a,b) of elements in G to an element (a*b) in G following

Axioms

- *Closure*: If a and b in G, then a * b is also in G
- *Associative*: a * (b * c) = (a * b) * c for all a,b,c in G
- *Identity*: Exists e in G, s.t. a*e = e*a = a for all a in G
- *Inverse*: for each a in G, exists a' in G, s.t. a*a' = a'*a = e

Finite group: finite number of elements

Groups, Rings, Fields

Abelian Group

— Group that satisfies

— *Commutative*: If a and b in G, then a * b = b * a

Rings

— A ring R, denoted by {R, +, x} is a set of elements with two binary operations: addition, multiplication. For all a,b,c in R the following axioms are satisfied

— R is **abelian group** with respect to the addition

— *Closure* under multiplication: ab in R

— *Associativity* of multiplication: a(bc) = (ab)c

— *Distributive laws*: a(b+c) = ab + ac

(a + b)c = ac + bc

Groups, Rings, Fields

Commutative Ring

- A **ring** that also satisfies
- *Commutativity of multiplication*: ab = ba in R

Integral Domain

- R is a commutative **ring** that also satisfies
- *Multiplicative identity*: exists **1**, s.t. a**1** = **1** a = a
- *No zero divisors*: ab = 0 , implies either a = 0 or b = 0

Field

- F is a field, { F, +, x } that satisfies
- F is an **integral domain**
- *Multiplicative inverse*: for each a in F, except 0, exists an element a^{-1}, s.t. $a(a^{-1}) = (a^{-1})a = 1$

Finite Fields GF(p)

Can write fields of the form GF(p), where p is prime

Addition and multiplication over GF(p) are mod p

Focus on p = 2

Example:

GF(2) addition: equivalent to XOR

multiplication: equivalent to AND

How to divide? Multiply by multiplicative inverse

Finding the multiplicative inverse

1.- Can look for a^{-1} such that $(a^{-1} . a) \equiv 1$

2.- Can use the extended Euclidean algorithm

Finite Fields - Applying GF(2) to NC

Example:

GF(2) addition: XOR

multiplication: AND

Given 2 data packets

P1: 01011001 P2: 10001001

calculate the content of the coded packet P1+P2.

What are the coefficients?

P1 + P2 = 01011001 (XOR bit by bit)

10001001

11010000

Modular Arithmetic

Modulus

— If a is an integer, n > 0 integer, we define a mod n to be the remainder when a is divided by n

— The integer n is called the modulus

— For any integer a, we can write

$$\mathbf{a = qn + r}, \text{ with } 0 \le r < n, \text{ and } q = \lfloor a/n \rfloor$$

— E.g., 11 mod 7 = 4 , -11 mod 7 = 3

Congruent modulo n

— If (a mod n) = (b mod n), and it's expressed

$$a \equiv b \pmod{n}$$

— E.g., $20 \equiv 6 \pmod{7}$

Modular Arithmetic

Properties of congruencies

— $a \equiv b \pmod{n}$ if $n|(a-b)$

— $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$

— If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$

Modular arithmetic operations

$[(a \bmod n) + (b \bmod n)] \bmod n = (a+b) \bmod n$

$[(a \bmod n) - (b \bmod n)] \bmod n = (a-b) \bmod n$

$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$

Rules of ordinary arithmetic involving addition, subtraction, multiplication carry over

Modular Arithmetic

Properties of modular arithmetic

— Define Z_n = { 0, 1, ..., n-1 } as the set of **residues** or **residue classes** mod n.

— Each element of Z_n is a residue class and can define it as

$$[\, j\,] = \{a : a \text{ is integer}, a \equiv j \pmod{n}\}$$

— Reducing k mod n: finding smallest non-negative integer a, such that $k \equiv a \pmod{n}$

Modular Arithmetic

Properties of modular arithmetic

— If $(a + b) \equiv (a + c) \pmod{n}$, then $b \equiv c \pmod{n}$

— If $(a \times b) \equiv (a \times c) \pmod{n}$, then $b \equiv c \pmod{n}$ if a is relatively prime to n, i.e., $\gcd(a,n) = 1$

Why is the last property important?

What happens when “p” is not a prime?

Will modular arithmetic still work?

Example 1:

If gcd(a,n) ≠ 1, the last equation does not hold

e.g. 6 x 3 = 18 = 2 mod 8

and 6 x 7 = 42 = 2 mod 8   but

3 mod 8 ≠ 7 mod 8

What happens when “p” is not a prime?

Will modular arithmetic still work?

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

x	0	1	2	3
0	0	0	0	0
1	0	**1**	**2**	**3**
2	0	**2**	**0**	**2**
3	0	**3**	**2**	**1**

What about GF(2^n)?

Since 2^n is not a prime, operations are defined in a different way
— polynomial arithmetic

Ordinary polynomial arithmetic:
— A polynomial of degree n
— $F(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 x^0 = \Sigma$ ai xi
— a_i are the coefficients, chosen from a set

Operations:

Addition $f(x) + g(x) = \Sigma (a_i + b_i) x_i$

Multiplication $f(x) \times g(x) = \Sigma C_i x^i$

with $c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0$

What about GF(2^n)?

Polynomial arithmetic in GF(2^n):

Arithmetic follows rules of polynomial arithmetic

Arithmetic of coefficients is performed modulo 2
- — i.e.,using GF(2) addition/multiplication for coefficients of the same order
- — e.g., (a_i + b_i) mod 2

If multiplication results in a polynomial greater than n-1, then the polynomial is reduced modulo an irreducible polynomial p(x)
- — Think of it as a mod p(x) operation: divide by p(x), keep the remainder

Example GF(2^2)

How about 2 +3?

$2 = (10)_b$ and $3 = (11)_b$

Thus, 2 + 3 becomes

$x + (x + 1) = 1$

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Irreducible polynomial

$p(x) = x^2 + x + 1$

(111)b

Example GF(2^2)

How about 2 x 3?

$x \ (x + 1) = (x^2 + x) \bmod p(x)$

$$
\begin{array}{r|l}
 & \quad 1 \\
\hline
x^2 + x + 1 & x^2 + x \\
 & x^2 + x + 1 \\
\hline
 & \qquad 1
\end{array}
$$

x	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Irreducible polynomial

$p(x) = x^2 + x + 1$

(111)b

How to implement multiplication GF(2^n)?

A.- Product (shifts+ XORs)

— 1) Pick one as multiplier (M) and another as multiplicand (m)

— 2) For each “1” in “M”, left shift the “m” by the position of the “1”

— 3) XOR shifted versions

B.- Modulo irreducible polynomial (long division)

— 1) Initialize: F(0) = Result of part A

— 2) Take irreducible polynomial p(x) left shift until first “1” of polynomial and of value F match

— 3) F(i+1) = (shifted p(x)) XOR (F(i))

— 4) Stop if first “1” of F(i+1) occurs in the (n-1)-th bit

How to implement multiplication GF(2^n)?

M = $(00010001)_b$ and m = $(10100111)_b$

```
       101001110000      (m shifted 4 times)
(XOR)      10100111      (m shifted 0 times)
       ____________
       101011010111
```

B.- Modulo irreducible polynomial

$p(x) = x^8 + x^4 + x^3 + x + 1 \quad (100011011)_b$

```
       101011010111   mod p(x)
(XOR)  100011011000
       ____________
       001000001111
(XOR)  001000110110
       ____________
       000000111001   → (00111001)b
```

Take Away Points

Use of finite fields necessary to maintain packet size

Addition is simple operation

Multiplication operation becomes more complex with larger field size

Why use (or not) look up tables?
- For small field size, it is efficient and requires small storage capability
- For larger field size, large look-up tables

fifi-python

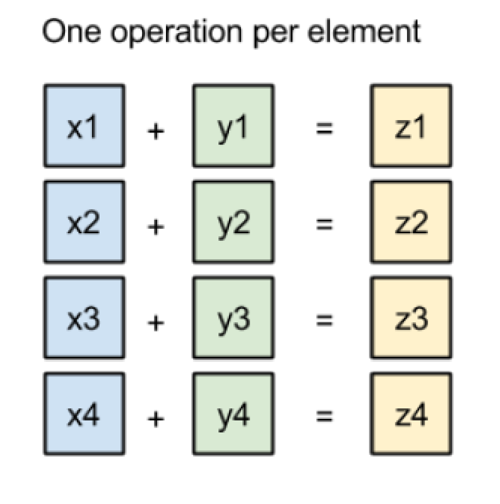
Input by morten v. pedersen

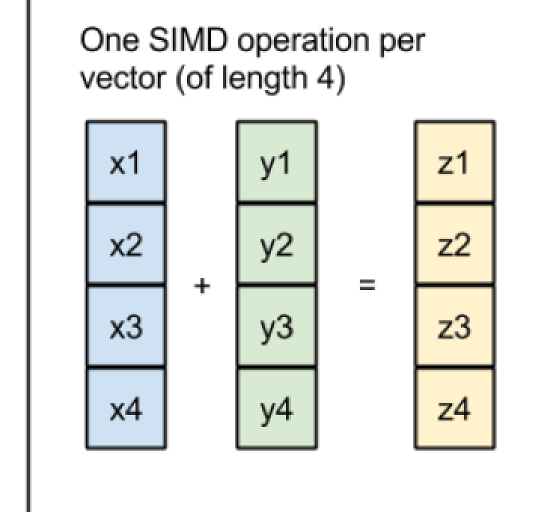
Finite field implementation

- Finite fields are a key component in many important applications:
  - — Cryptography.
  - — Error/Erasure correcting codes.
  - — Error detection e.g. CRC (Cyclic Redundancy Checks).
- Having an efficient implementation is important to evaluate system critical parameters:
  - — Throughput, latency, energy consumption, memory consumption etc.
- Today we will use the fifi-python library to perform finite field computations.

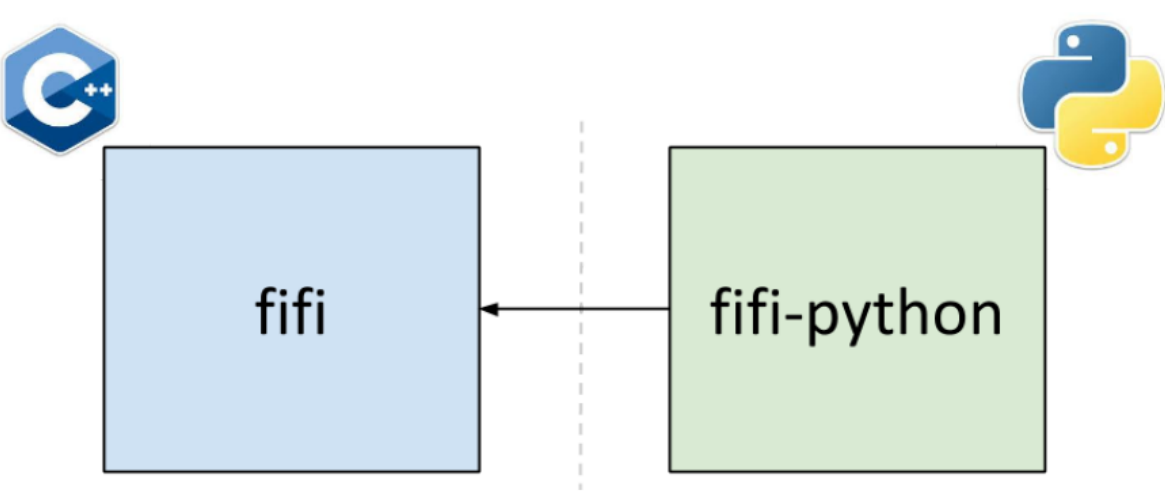
A note on performance

- In the past few year performance has increase significantly
- This has happened because we understand how to utilize specialized instructions on modern CPUs called SIMD (Single Instruction Multiple Data).
- fifi and fifi-python uses this extensively (on both x86 and ARM CPUs)





About fifi and fifi-python




```
fifi-python/
├── config.py
├── examples
│   ├── fifi_simple_api.py
│   ├── fifi_simple_api_table_example.py
│   ├── hello_fifi.py
│   └── hello_simple_api.py
├── LICENSE.rst
├── NEWS.rst
├── README.rst
├── src
│   └── fifi_python
├── test
│   ├── data.json
│   └── fifi_python_tests.py
├── waf
└── wscript
```

Exploring the API

- To use the library `import` it into your scripts.

In [ ]:
```
import fifi
```

- You can explore the API using the built-in `help()` function

In [2]:
```
help(fifi)
```

```
Help on module fifi:

NAME
    fifi

FILE
    /home/mvp/Dropbox/work_code/notebooks/slides_fifi_python/fifi.so

CLASSES
    Boost.Python.instance(__builtin__.object)
        extended_log_table_binary16
        extended_log_table_binary4
        extended_log_table_binary8
        full_table_binary4
        full_table_binary8
        log_table_binary16
        log_table_binary4
        log_table_binary8
        optimal_prime_prime2325
        simple_online_binary
        simple_online_binary16
        simple_online_binary4
        simple_online_binary8

    class extended_log_table_binary16(Boost.Python.instance)
     |  A finite field implementation
     |
```

Using help()

- You can also get more specific help, by passing a specific class or function.

```
In [4]: import fifi
        help(fifi.simple_online_binary4.add)

Help on method add:

add(...) unbound fifi.simple_online_binary4 method
    Returns the sum of two field elements.

            :param a: The augend.

            :param b: The addend.
            :returns: The sum of a and b.
```

A small example

In [8]:

```
import fifi

field = fifi.simple_online_binary4()
order = 2**4

table = ''


for i in range(order):
    for j in range(order):
        table += '{:02d} '.format(field.multiply(i,j))
    table += '\n'

print(table)
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 02 04 06 08 10 12 14 03 01 07 05 11 09 15 13
00 03 06 05 12 15 10 09 11 08 13 14 07 04 01 02
00 04 08 12 03 07 11 15 06 02 14 10 05 01 13 09
00 05 10 15 07 02 13 08 14 11 04 01 09 12 03 06
00 06 12 10 11 13 07 01 05 03 09 15 14 08 02 04
00 07 14 09 15 08 01 06 13 10 03 04 02 05 12 11
00 08 03 11 06 14 05 13 12 04 15 07 10 02 09 01
00 09 01 08 02 11 03 10 04 13 05 12 06 15 07 14
00 10 07 13 14 04 09 03 15 05 08 02 01 11 06 12
00 11 05 14 10 01 15 04 07 12 02 09 13 06 08 03
00 12 11 07 05 09 14 02 10 06 01 13 15 03 04 08
00 13 09 04 01 12 08 05 02 15 11 06 03 14 10 07
00 14 15 01 13 03 02 12 09 07 06 08 04 10 11 05
00 15 13 02 09 06 04 11 01 14 12 03 08 07 05 10
```

Hands-On

Make sure fifi-python is installed (lecture slides)

Create a script (e.g. print_table.py) and write the code to print both the multiplication and division tables.

Run the script python print_table.py.

Using the Simple API

- The previous example use the field object directly.
- It is also possible to standard arithmetic operations (+,-,*,/,~) with field elements.
- To use it copy the fifi-python/examples/fifi_simple_api.py to your scripts folder.

In [3]:

```
#! /usr/bin/env python

# Copyright Steinwurf ApS 2011-2013.
# Distributed under the "STEINWURF RESEARCH LICENSE 1.0".
# See accompanying file LICENSE.rst or
# http://www.steinwurf.com/licensing

from fifi_simple_api import B4


def main():
    a = B4(13)
    b = B4(7)

    print("{a} + {b} = {result}".format(a=a, b=b, result=a + b))
    print("{a} - {b} = {result}".format(a=a, b=b, result=a - b))
    print("{a} * {b} = {result}".format(a=a, b=b, result=a * b))
    print("{a} / {b} = {result}".format(a=a, b=b, result=a / b))
    print("~{a} = {result}".format(a=a, result=~a))

if __name__ == '__main__':
    main()
```

```
13 + 7 = 10
13 - 7 = 10
13 * 7 = 5
13 / 7 = 8
~13 = 4
```