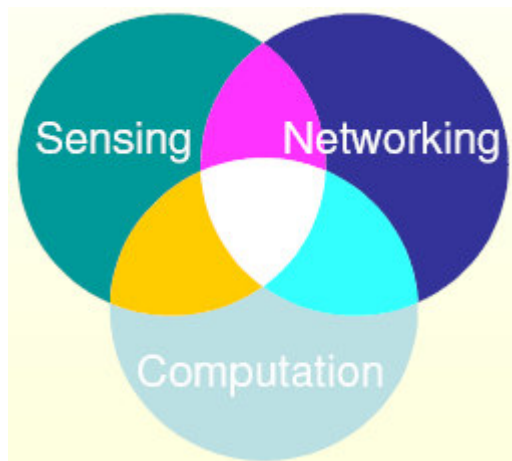


# Coding and its applications in sensor networks

Jie Gao

Computer Science Department  
Stony Brook University



# Paper

- **[Dubois05]** Henri Dubois-Ferriere, Deborah Estrin and Martin Vetterli, [Packet Combining in Sensor Networks](#), Sensys'05.
- **[Dimakis05]** A. G. Dimakis, V. Prabhakaran and K. Ramchandran, [Ubiquitous Access to Distributed Data in Large-Scale Sensor Networks through Decentralized Erasure Codes](#), Symposium on Information Processing in Sensor Networks (IPSN'05), April, 2005.

# Source coding

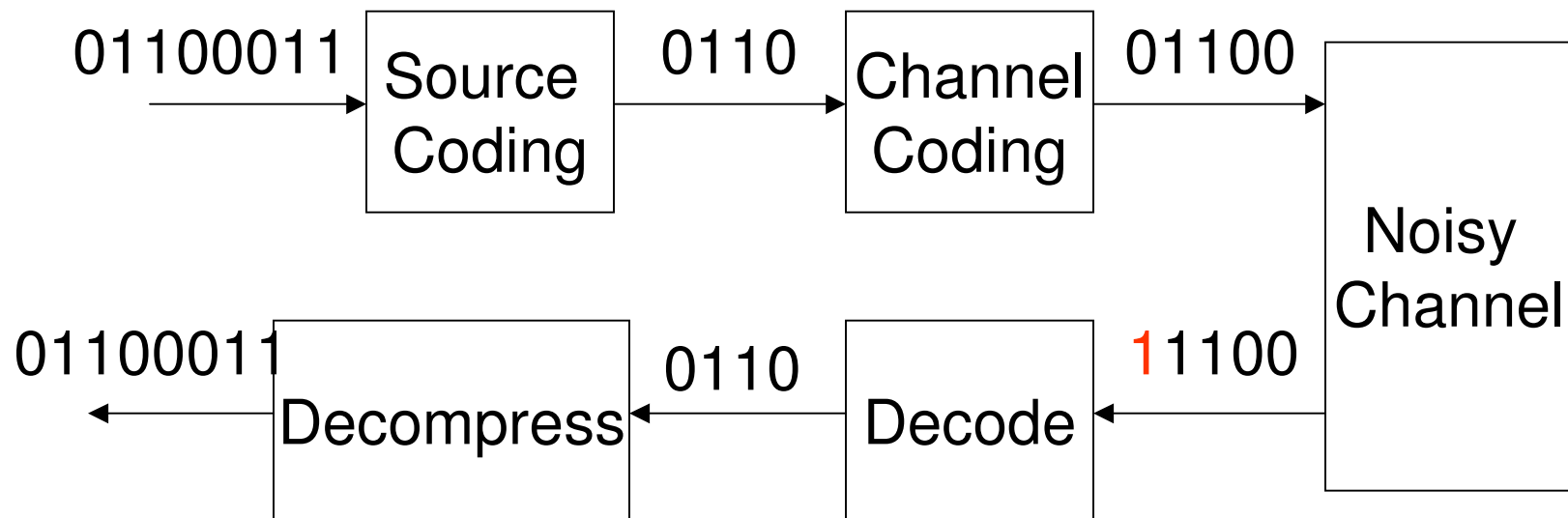
- Compression.
- What is the minimum number of bits to represent certain information? What is a measure of information?
- Entropy, Information theory.

# Channel coding

- Achieve fault tolerance.
- Transmit information through a noisy channel.
- Storage on a disk. Certain bits may be flipped.
- Goal: recover the original information.
- How? duplicate information.

# Source coding and Channel coding

- Source coding and channel coding can be separated without hurting the performance.



# Coding in sensor networks

- Sensors generate too much data.
- Nearby sensor readings are correlated.
- Sensor networks are not reliable.
- Nodes may die, links may fail, nodes may be compromised.
- Corrupted messages by a noisy channel.
- Communication failures.
- Node failures – fault tolerance storage.
- Adversary inject false information.

# Channels

- The media through which information is passed from a sender to a receiver.
- **Binary symmetric channel**: each symbol is flipped with probability  $p$ .
- **Erasure channel**: each symbol is replaced by a “?” with probability  $p$ .
- We first focus on binary symmetric channel.

# Encoding and decoding

- **Encoding:**
- Input: a string of length  $k$ , “data”.
- Output: a string of length  $n > k$ , “codeword”.
  
- **Decoding:**
- Input: some string of length  $n$  (might be corrupted).
- Output: the original data of length  $k$ .

# Error detection and correction

- Error detection: detect whether a string is a valid codeword.
- Error correction: correct it to a valid codeword.
- **Maximum likelihood Decoding**: find the codeword that is “closest” in Hamming distance, I.e., with minimum # flips.
- How to find it?
- For small size code, store a codebook. Do table lookup.
- NP-hard in general.

# Scheme 1: repetition

- Simplest coding scheme one can come up with.
- Input data: 0110010
- Repeat each bit 11 times.
- Now we have
- 000000000001111111111111111111111100000000  
00000000000000011111111111000000000000
- Decoding: do majority vote.
- Detection: when the 10 bits don't agree with each other.
- Correction: 5 bits of error.

## Scheme 2: Parity-check

- Add one bit to do parity check.
- Sum up the number of “1”s in the string. If it is even, then set the parity check bit to 0; otherwise set the parity check bit to 1.
- Eg. 001011010, 111011111.
- Sum of 1's in the codeword is even.
- 1-bit parity check can detect 1-bit error. If one bit is flipped, then the sum of 1s is odd.
- But can not detect 2 bits error, nor can correct 1-bit error.

# More on parity-check

- Encode a piece of data into **codeword**.
- Not every string is a codeword.
- After 1 bit parity check, only strings with even 1s are valid codeword.
- Thus we can detect error.
  
- Minimum Hamming distance between any two codewords is 2.
- Suppose we make the min Hamming distance larger, then we can detect more errors and also correct errors.

# Scheme 3: Hamming code

- Intuition: generalize the parity bit and organize them in a nice way so that we can detect and correct more errors.
- Lower bound: If the minimum Hamming distance between two code words is  $k$ , then we can detect at most  $k-1$  bits error and correct at most  $\lfloor k/2 \rfloor$  bits error.
- Hamming code (7,4): adds three additional check bits to every four data bits of the message to correct any single-bit error, and detect all two-bit errors.

# Hamming code (7, 4)

- Coding: multiply the data with the encoding matrix.

$$H_e := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

- Decoding: multiply the codeword with the decoding matrix.

$$H_d := \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

# An example: encoding

- Input data: 
$$\mathbf{p} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

- Codeword: 
$$H_e \mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \mathbf{r}$$

Systematic code: the first k bits is the data.

Original data is preserved

# An example: decoding

- Decode:

$$H_d \mathbf{r} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

- Now suppose there is an error at the  $i$ th bit.

- We received

$$\mathbf{r} + \mathbf{e}_i$$

- Now decode:

$$H_d(\mathbf{r} + \mathbf{e}_i) = H_d \mathbf{r} + H_d \mathbf{e}_i$$

$$H_d \mathbf{r} + H_d \mathbf{e}_i = \mathbf{0} + H_d \mathbf{e}_i = H_d \mathbf{e}_i$$

- This picks up the  $i$ th column of the decoding vector!

# An example: decoding

- Suppose
$$\mathbf{s} = \mathbf{r} + \mathbf{e}_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$
- Decode:
$$H_d \mathbf{s} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Second bit is wrong!
- Data more than 4 bits? Break it into chunks and encode each chunk.

# Linear code

- Most common category.
- Succinct specification, efficient encoding and error-detecting algorithms – simply matrix multiplication.
- Code space: a linear space with dimension  $k$ .
- By linear algebra, we find a set of basis

$$\mathbf{x}_1, \dots, \mathbf{x}_k \subseteq \mathbb{F}_q^n$$

- Code space:

$$C = \left\{ \sum_{i=1}^k \alpha_i \cdot \mathbf{x}_i \mid \alpha_1, \dots, \alpha_k \in \mathbb{F}_q \right\}$$

- Generator matrix  $\mathbf{G} \in \mathbb{F}_q^{k \times n}$

$$C = \{ \alpha \mathbf{G} \mid \alpha \in \mathbb{F}_q^k \}$$

# Linear code

- Null space of dimension  $n-k$ :

$$\mathbf{H}^T \in \mathbb{F}_q^{(n-k) \times n}$$

- Parity check matrix.

$$C = \{\mathbf{y} \mid \mathbf{y}\mathbf{H} = \mathbf{0}\}$$

- Error detection: check

$$\mathbf{y}\mathbf{H} \neq \mathbf{0}$$

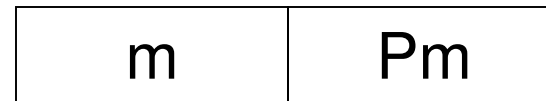
- Hamming code is a linear code on alphabet  $\{0,1\}$ . It corrects 1 bit and detects 2 bits error.

# Linear code

- A linear code is called systematic if the first  $k$  bits is the data.
- Generation matrix  $G$ :

$I_{k \times k}$	$P_{k \times (n-k)}$
------------------	----------------------

- If  $n=2k$  and  $P$  is invertible, then the code is called invertible.



- A message  $m$  maps to
- Parity bits can be used to recover  $m$ .
- Detect more errors? Bursty errors?

Parity bits

# Reed Solomon codes

- Most commonly used code, in CDs/DVDs.
- Handles bursty errors.
- Use a large alphabet and algebra.
- Take an alphabet of size  $q > n$  and  $n$  distinct elements  $\alpha_1, \dots, \alpha_n \in F_q$
- Input message of length  $k$ :  $c_0, \dots, c_{k-1}$
- Define the polynomial  $C(x) \stackrel{\text{def}}{=} \sum_{j=0}^{k-1} c_j x^j$
- The codeword is  $\langle C(\alpha_1), \dots, C(\alpha_n) \rangle$

# Reed Solomon codes

- Rephrase the encoding scheme.
- Unknowns (variables): the message of length  $k$

$$c_0, \dots, c_{k-1}$$

- What we know: some equations on the unknowns.

$$\langle C(\alpha_1), \dots, C(\alpha_n) \rangle \quad C(x) \stackrel{\text{def}}{=} \sum_{j=0}^{k-1} c_j x^j$$

- Each of the coded bit gives a linear equation on the  $k$  unknowns.  $\rightarrow$  A linear system.
- How many equations do we need to solve it?
- We only need length  $k$  coded information to solve all the unknowns.

# Reed Solomon codes

- Write the linear system by matrix form:

$$\begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \alpha_1^{k-1} \\ 1 & \alpha_2 & \alpha_2^2 & \alpha_2^{k-1} \\ \dots & \dots & \dots & \dots \\ 1 & \alpha_k & \alpha_k^2 & \alpha_k^{k-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \dots \\ c_{k-1} \end{bmatrix} = \begin{bmatrix} C(\alpha_1) \\ C(\alpha_2) \\ \dots \\ C(\alpha_k) \end{bmatrix}$$

- This is the Edmond matrix. So it's invertible.
- This code can tolerate n-k errors.
- **Any** k bits can recover the original message.

# Coding in communication

# Coding in networks

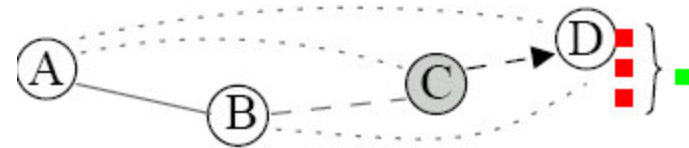
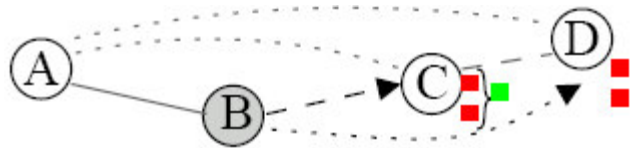
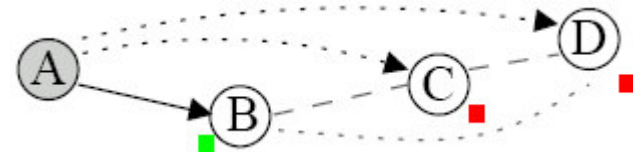
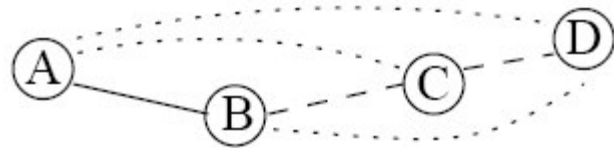
- On Internet, coding and decoding are handled at end nodes.
- Sender wraps up the gift (encoding).
- Routers only forward packets, never check what is inside or whether it's corrupted or not.
- Receiver opens the box and checks whether the gift is broken or not (decoding).
- Reliability is achieved by re-transmission.

# Coding in sensor networks

- End-to-end re-transmission is too costly.
- Target scenario:
  - low-rate network utilization.
  - Packet loss is mainly caused by fading and attenuation, rather than congestion and collisions.
  - Corruption consists of small errors rather than long burst errors.
- Buffer corrupted messages.
- Combine and recover from multiple corrupted messages.
- Overhear.

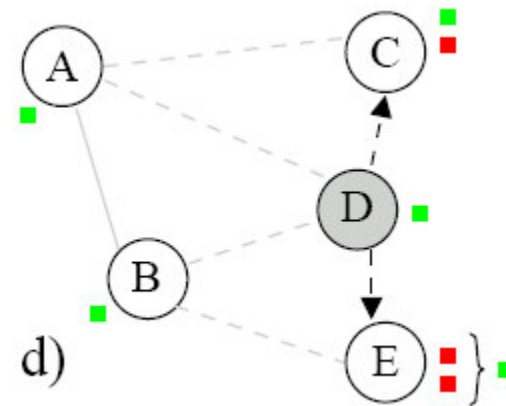
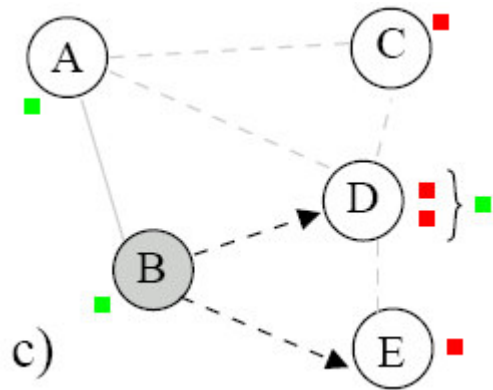
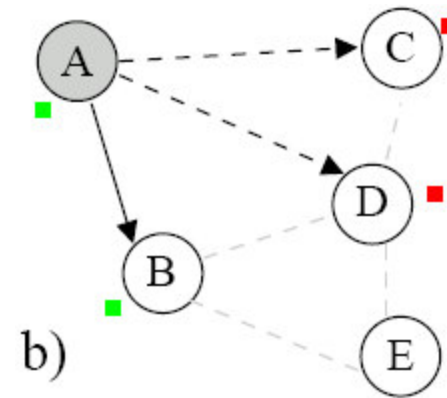
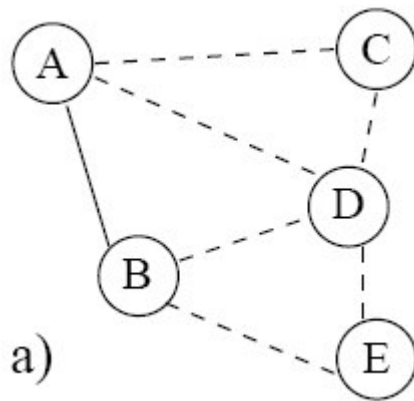
# Packet combining

- A multi-hop scenario.



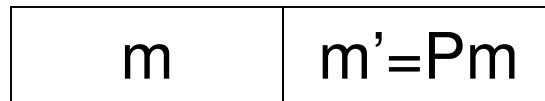
# Packet combining

- A broadcast scenario



# How to combine corrupted messages?

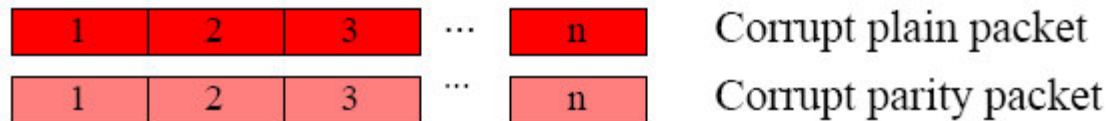
- Simplest one: use repetition code.
- Use majority vote from multiple (corrupted) copies.
- Use an invertible systematic linear code.
- Each message  $m$  is encoded into



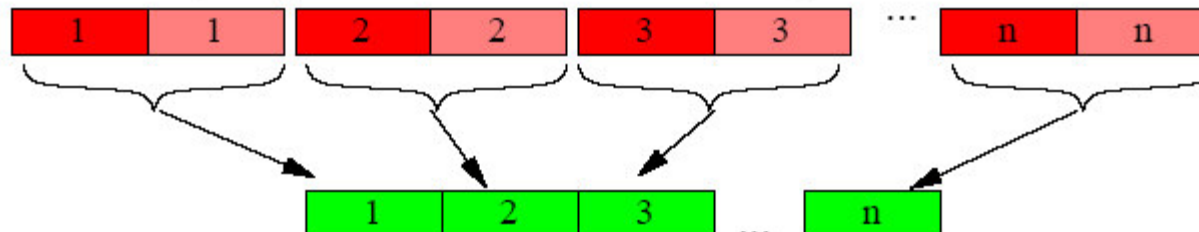
Plain packet    Parity packet

- A parity packet can recover the message.  $m = P^{-1}m'$ .
- A (corrupted) plain packet + a (corrupted) parity packet can decode for the original data.

# Decoding



Plain and parity blocks are interleaved and passed to the decoder.



# How to combine corrupted messages?

- We can send both plain and parity packets. But the rate is decreased (we use twice the amount of bits to send the information).
- Idea: use broadcast property.
- Send either plain packet or parity packet.
- If the packet is not corrupted then we are fine.
- If we get two corrupted packets, then we recover.

$m$	Action
Plain	None
Parity	Invert

a)

$m$	$n$	Action
Plain	Plain	Merge
Plain	Parity	Decode
Parity	Plain	Decode
Parity	Parity	Merge, invert

b)

# Packet merging

- What if we receive two packets of the same type?
- We can use majority vote if we have 3 or more corrupted packets.
- We also attach 1 bit parity check.
- For 2 packets  $m_1$ ,  $m_2$ , we take the XOR.
- The bits with 1 are possible errors.
- If there are few errors (1 or 2), try do some flipping and check the parity sum.

# Packet merging and decoding

- We hope to be able to decode instead of merge.
- We want two corrupted messages are of different types.
- 1-hop retransmission: alternates between plain and parity.
- Multi-hop: the initial transmission is of opposite type to the last received packet.
- Flooding: randomly choose between parity and plain.

# Coding in storage

# Use coding for fault tolerance

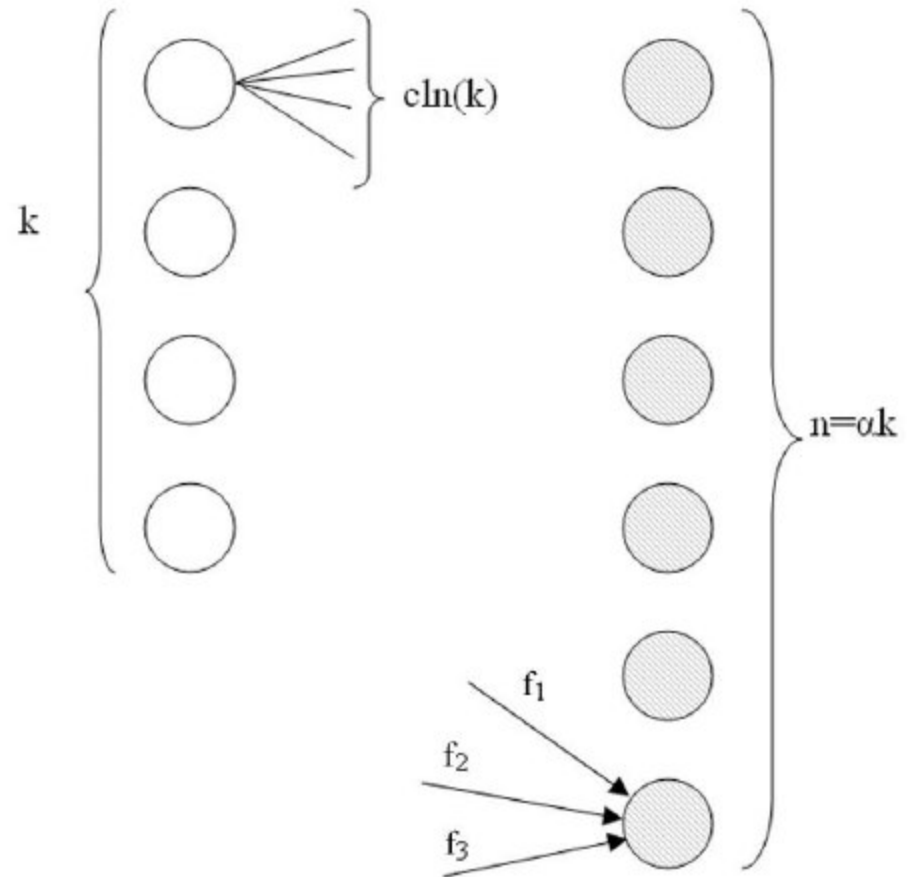
- If a sensor die, we lose the data.
- For fault tolerance, we have to duplicate data s.t. we can recover the data from other sensors.
- Straight-forward solution: duplicate it at other places.
- Storage size goes up!
  
- Use coding to keep storage size as the same.
- What we pay: decoding cost.

# Problem setup

- Setup: we have  $k$  data nodes, and  $n > k$  storage nodes (data nodes may also be storage nodes).
- Each data node generates one piece of data.
- Each storage node only stores one piece of (coded) data.
- We want to recover data by using **any**  $k$  storage nodes.
  
- Sounds familiar? Reed Solomon code.
- But it is centralized -- we need all the  $k$  inputs to generate the coded information.

# Distributed random linear code

- Each node sends its data to  $m=O(\ln k)$  random storage nodes.
- A storage node may receive multiple pieces of data  $c_1, c_2, \dots, c_k$ , but it stores a random combination of them. E.g.,  $a_1c_1 + a_2c_2 + \dots + a_kc_k$ , where  $a$ 's are random coefficients.

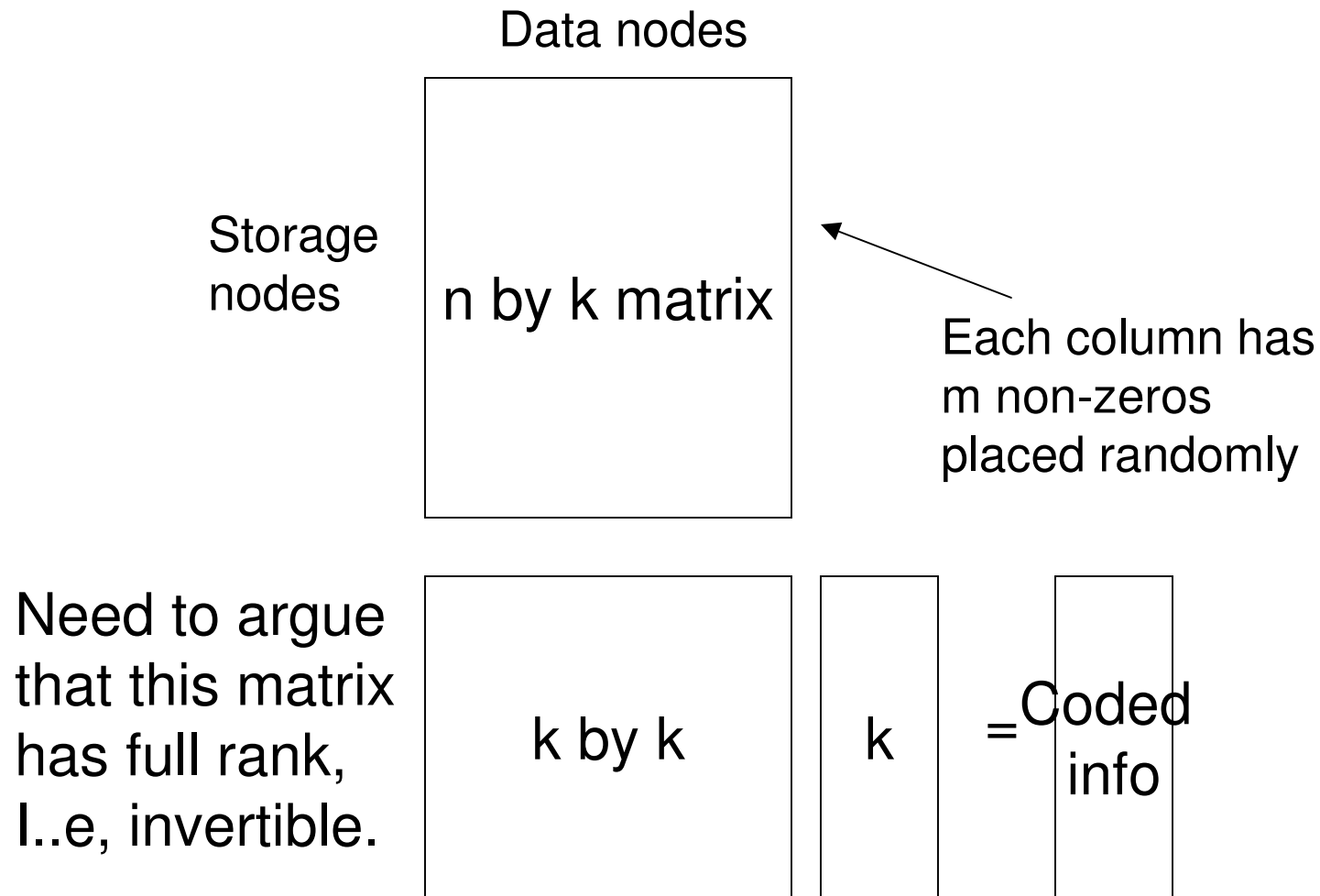


# Coding and decoding

- Storage size keeps almost the same as before.
- The random coefficients can be generated by a pseudo-random generator. Even if we store the coefficients, the size is not much.
- Claim: we can recover the original  $k$  pieces of data from any  $k$  storage nodes.
- Think of the original data as unknowns (variables).
- Each storage node gives a linear equation on the unknowns  $a_1c_1+a_2c_2+\dots+a_kc_k = s$ .
- Now we take  $k$  storage nodes and look at the linear system.

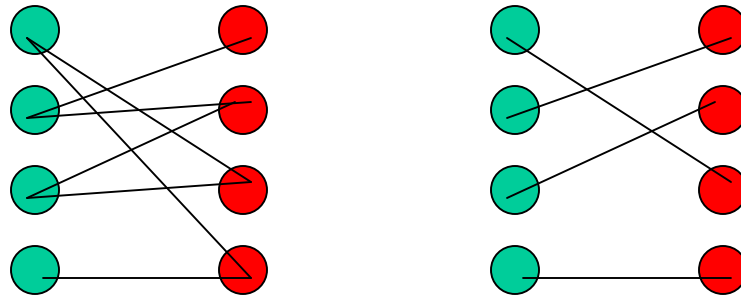
# Coding and decoding

- Take arbitrary  $k$  storage nodes.



# Main theorem

- A bipartite graph  $G=(X, Y)$ ,  $|X|=k$ ,  $|Y|=k$ .
- $X$ : the data nodes;  $Y$ : the  $k$  storage nodes.



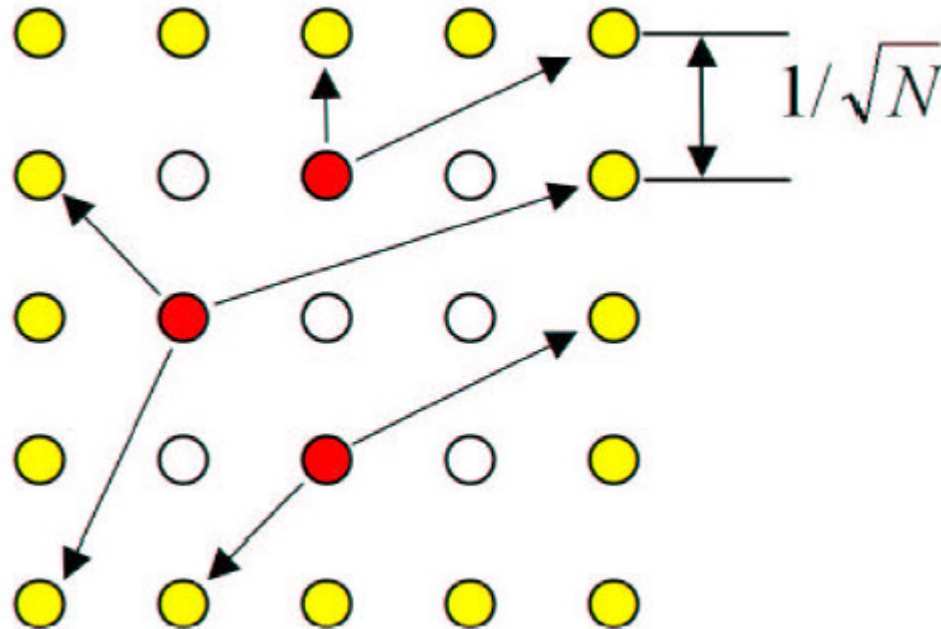
- Edmond's theorem: the matrix has full rank if the bipartite graph has a **perfect matching**.
- Now, we only need to show that the bipartite graph  $G$  has a perfect matching with high probability.

# Main theorem

- Upper bound: if storage node picks  $O(\ln k)$  storage randomly, the bipartite graph  $G$  has a perfect matching with high probability.
- Lower bound:  $\Omega(\ln k)$  is necessary.
- Proof:
  - Any storage node has to have at least one piece of data.
  - Otherwise, the matrix has a zero row!
  - Throw data randomly to cover all the storage nodes.
  - Coupon collector problem: each time get a random coupon. In order to collect all  $n$  different types of coupon, with high probability one has to get in total  $\Omega(n \ln n)$  coupons.

# Perimeter storage

- Potential users outside the network have easy access to perimeter nodes; Gateway nodes are positioned on the perimeter.



# Pros and Cons

- No extra infrastructure, only a point-to-point routing scheme is needed.
- Robust to errors – just take  $k$  good copies.
- Fault tolerance – sensors die? Fine...
- No centralized processing, no routing table or global knowledge of any sort.
- Very resilient to packet loss due to the random nature of the scheme.
- Achieves certain data privacy. If the coding scheme (the random coefficients) is kept from the adversary, the adversary only sees random data.

# Pros and Cons

- Information is coded, in other words, scrambled.
- Have to decode the whole  $k$  pieces, even only 1 piece of data is desired.
- Doesn't explore locality – usually we don't go to arbitrary  $k$  storage nodes, we go to the closest  $k$  nodes.

# Summary

- Combing coding idea with sensor storage and communication schemes is a very promising area.
- Distributed coding schemes.
- Locality-aware (geometry-aware) coding schemes.