Lecture 13: Coding, Error Detection and Correction

Errors and Losses

- **Physical layers use encoding to protect link layer from chip errors**
- **All or nothing: if chip errors exceed layer 1 robustness, you lose the whole packet (bad CRC)**
- **We can use these techniques at higher layers as well: *erasure coding***
  - Encoding data of length L as k symbols: any n of the k symbols can regenerate the original data ($n \geq L$).

Cyclic Redundancy Check (CRC), revisited

- **Distill n bits of data into a c bit CRC, $c << n$**
- **Can't detect all errors (2^{-c} chance another packet's CRC matches)**
- **CRCs are designed to detect certain forms of errors more than others**
- **Stronger than checksums; detect**
  - A message with any 1 or 2 bits in error
  - A message with any odd number of errors
  - A message with an error burst as wide as the CRC

Message Authenticaion Codes

- **When sending packets securely (confidentially), sometimes you use a Message Authentication Code (MAC) instead of a CRC**
- **Kind of like a CRC, but seeded with a secret**
- **MAC needs different properties to be strong against attackers**
  - If any bit in the message changes, each bit in the MAC needs to have an equal probability of being 0 or 1
  - Starting with packet P, can't append data and generate new MAC
  - *Can flip a bit and get the same MAC value*

Cyclic Redundancy Check, continued

- **Can be computed iteratively (e.g., as a packet is spooled out)**
- **Mathematical basis**
  - CRC of length n computed as an nth degree polynomial
  - CRC of length n is remainder after dividing message by a number $k > 2^n$
  - $k = 25$, $k = 11001$, $x^4 + x^3 + x^0$
- **To detect burst errors of length b, $n > b$**

Fields

- **A set of numbers of which you can add, subtract, multiply, and divide**
- **A field can be finite (Galois Field)**
- **E.g., Galois Field $GF(p)$ field of integers modulo p, p is prime**
- **In $GF(7)$, $6 + 3 = 2$**

Reed-Solomon

- **Standard erasure coding technique: used in CDs**
- **Core idea: any k distinct data points define a unique polynomial of degree $k-1$**
- **Data to transmit defines the polynomial P**
- **Compute coded data $C = P(x)$ for x_0, x_1, x_n**
- **Transmit C**
- **A receiver that gets k different x_n values can reconstitute original polynomial (and data)**

Power of Erasure Coding

- **Make data $k-1$ term polynomial**
- **Compute n data points, where $n > k$**
- **Any k data points can successfully regenerate data**
- ***Can tolerate $n-k$ losses, or $\frac{n-k}{2}$ errors***
- **Known errors/losses are called erasures: erasure coding**

Code Word Sizes

- **Large polynomials lead to large values: $63 \cdot 19^{10}$!**
- **Reed-Solomon works on fields**
- **GF(256),** $63 \cdot 19^{10} = 106$
- **Reed-Solomon uses GF(256);**
- **Sends 223 data symbols, 32 symbols are parity values (223,255)**
- **8 bits can have very efficient implementations**

Error Burstiness

- **Reed-Solomon divides data into code symbols (e.g., 8 bits)**
- **Any one bit error invalidates that symbol**
- **Robust to bursts of errors, weaker against random errors**
- **Big bursts can overwhelm the code: CDs use Cross-Interleaved Reed-Solomon Coding (CIRC) to spread errors across symbol blocks**

Reed-Solomon

- **Useful for making media more resilient: operates on small datum**
- **Increasing the coding factor k increases robustness: can be made robust to arbitrary losses**
- **Is there some way to do something similar on large data (e.g., packets?)**

Erasure Codes, Revisited

- **Data of length k**
- **Erasure codes can regenerate data from any $k(1+\gamma)$ code symbols**
- **Do not necessarily handle errors**
- **Perfect world, $\gamma = 0$ (not possible in practice)**
- **Also want decoding to be fast/simple**

Problem Statement

- **Have a large piece of data**
- **Want to deliver it to many client nodes**
- **Do not want to maintain per-client state**
- **Goal: send packets, any k packets can reconsitute data with very high probability**
- **Example: television over IP**

Why Not Reed-Solomon

- **We're operating on a very large piece of data**
- **Can deal with larger chunks than Reed-Solomon**
- **Allows us to do something simpler?**

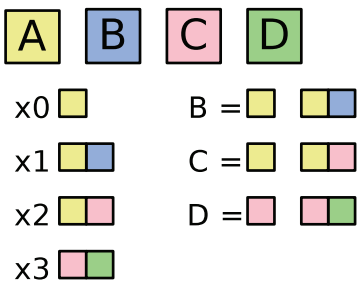
LT Codes

- **Data of length N**
- **Code words of length l, any l will do!**
- **Break data into $k = \lceil \frac{D}{L} \rceil$ chunks C**
- **Each transmitted packet C_i is an XOR of some number of random chunks**
- **Symbol generation is $O(\ln(k/\delta))$ on average**
- **Requires $k + O(\sqrt{k}\ln^2(k/\delta))$ symbols to decode with probability $1 - \delta$**

Power of XOR

- **Data A, B, C, D**
- **Have $x_0 = A, x_1 = A \oplus B, x_2 = A \oplus C, x_3 = C \oplus D$**
- $A = x_0$
- $B = A \oplus x_1$
- $C = A \oplus x_2$
- $D = C \oplus x_3$

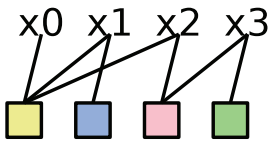
XOR, Visually



How to XOR?

- **How LT codes XOR the data into code words is critical**
- **$d(C_i)$ is the degree of codeword i: how many data items are XORed in it**
- **Example bad distributions**
  - $\forall C_i, d(C_i) = 1$: sending data in the clear
  - $\forall C_i, d(C_i) = k - 1$: each codeword is the XOR of all but one data symbol
  - Both have k unique code words, and require receiving all k

Codeword Degree



LT Decoding Algorithm

- **We receive n codewords**
- **Initially, all input symbols are *uncovered***
- **Each codeword with $d = 1$ *covers* the input symbol**
- **Ripple: set of covered input symbols that have not been processed**
- **For each symbol in the ripple**
  - Scan across the codewords and remove it from those that have it (via XOR), reducing degree by 1
  - If codeword has degree 1, cover its input symbol

LT Decoding

- **Decoding stops when the ripple is size 0**
  - If all input symbols are covered, success
  - If any input sybmols are uncovered, failure
- **How big should the ripple be?**
  - Small, so there is low redundancy in coverage
  - Never zero, because that halts decoding
- **Control ripple size through degree distribution ρ**

2-minute break

LT Encoding

- **Two goals**
  - Minimize number of needed encoding symbols (bps)
  - Minimize average degree of encoding symbols (CPU cycles)
- **Step through three examples:**
  - All-at-once: $\rho(1) = 1$
  - Ideal: $\rho(1) = 1/k, \rho(i) = 1/i(i-1)$
  - Robust: more on this later

All-at-once

- **How many encoded symbols does it take?**
- **Bins and balls: how many balls must be thrown to ensure that there is one in each of k bins?**
- $O(k \cdot \ln(k/\delta))$ **for probability** $1 - \delta$
- **Requires** $\ln(k/\delta)$ **factor (on average** $\ln(k)$**)**
- $k = 1000, \delta = 0.01\%, \ln(10^7) = 16$

Ideal distribution

- $\rho(1) = 1/k$, $\forall i, i = 2, \ldots, k, \rho(i) = 1/i(i-1)$
- **Expected behavior is perfect: one input has a d of 1, each cover releases another symbol**
- **Requires k symbols, sum of degrees is $k \cdot \ln(k)$**
- **Same sum of degrees as all-at-once, but only k symbols**

The World Is Not Ideal

- **"However, this heuristic analysis makes the completely unrealistic assumption that the expected behavior is the actual behavior, and this is far from the truth. In fact, the Ideal Soliton distribution works very poorly in practice because he expected size of the ripple is one, and even the smallest variance causes the ripple to vanish and thus the overall process fails to cover and process all input symbols." (Luby)**

Robust Distribution

- **Size of ripple behaves like a random walk**
- **Goal: make median point of random walk large enough to survive variations ($ln(k/\delta)\sqrt{k}$)**
- **Define a constant R, $R = c \cdot \ln(k/\delta)\sqrt{k}$**

$$\tau(i) = \begin{cases} R/ik & i = 1, \ldots, k/R - 1 \\ R\ln(R/\delta)/k & i = k/R \\ 0 & i = l/R + 1, \ldots, k \end{cases}$$

- **Add τ to ρ of Ideal, normalize**

Improvements of Robust Distribution

- **τ skews Ideal towards smaller degrees**
- **Requires $k + O(\sqrt{k} \cdot \ln^2(k\delta))$ (additive factor!)**
- **$k = 1000, \delta = 0.01\%$, requires an extra 88 symbols, 8% overhead.**
- **Average degree is still $\ln(k/\delta)$**

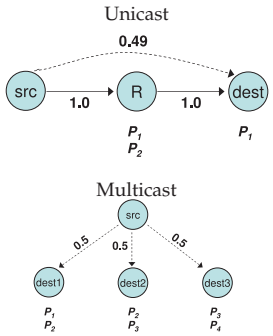
Implications

- **Example: server wants to deliver 1MB image (1000 1kB packets)**
- **Server generates packets from robust distribution**
- **No matter the properties of the lossy channel (burstiness, periodicity, etc.), there is a 99.99% chance that a client will be able to regenerate the image after 1088 packets**
- **Uses only $\ln(k/\delta)$ (e.g., 16) XOR operations per packet**
- **Hence the name, "Fountain Code"**

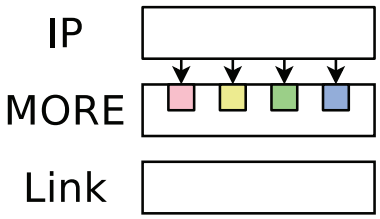
MORE

- **Chachulski et al., SIGCOMM 2007.**
- **"MAC-independent Opportunistic Routing & Encoding"**
- **Uses network coding in wireless networks to improve throughput**
- **Works with unicast flows as well as multicast!**
- **Sachin Katti, starting in EE/CS in January, co-authored**

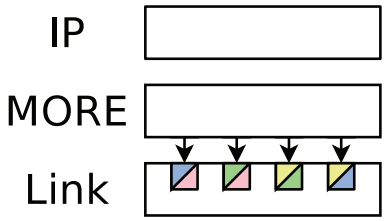
Motivation



MORE at Layer 2.5



MORE at Layer 2.5



MORE Summary

- A coded packet is a linear combination of native packets
- MORE adds a header that contains the linear coefficients
- MORE uses opportunistic reception: nodes broadcast coded packets to some number of next hops, more than one of which can forward the packet
- Source collects native packets into a batch, starts sending linear combinations of those packets, stops when it receives an end-to-end ACK.

Opportunistic Reception

- MORE maintains ETX-based route estimates
- MORE picks n forwarders which are closer to the destination than it, ordered according to their proximity (n closest)

Forwarding Packets

- Some received coded packets are innovative: they are linearly independent of prior packets (bounded by batch size K)
- Forwarders discard packets that are not innovative
- Whenever a forwarder receives a packet, it gets a transmit credit
- Sending a packet costs a credit
- ETX ordering prevents loops and credit explosion

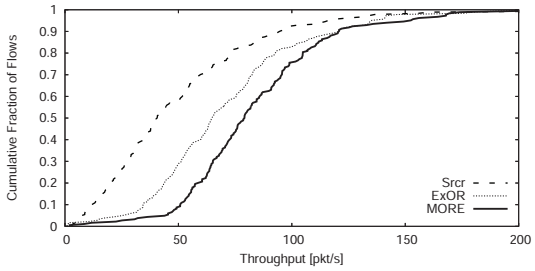
Encoding and Decoding Packets

- **Source makes each encoded packet a random linear combination of native packets**
- **Forwarders send random linear combinations of encoded packets (which are still linear combinations)**
- **K linearly independent combinations allow the destination to decode packet through simple matrix inversion.**

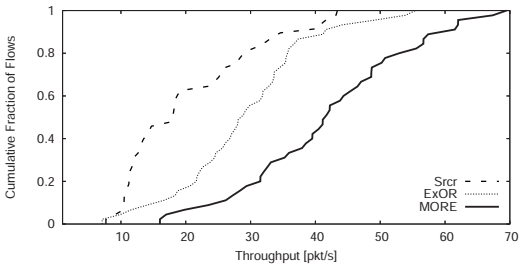
When To Stop?

- **When destination determines it can decode, it immediately sends an end-to-end ACK (don't even wait for decoding)**
- **Every node that hears the ACK stops**
- **When the source stops, forwarders stop getting transmit credits**
- **ACK packets are put on fast path: take precedence over all data packets**

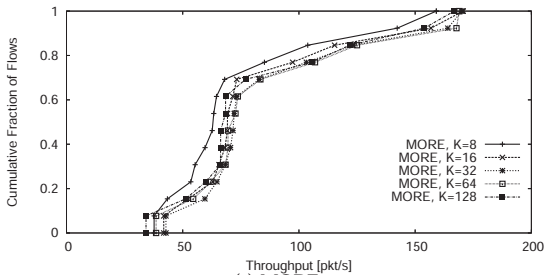
Improvement



Long Routes

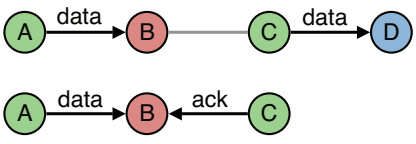


Batch Size

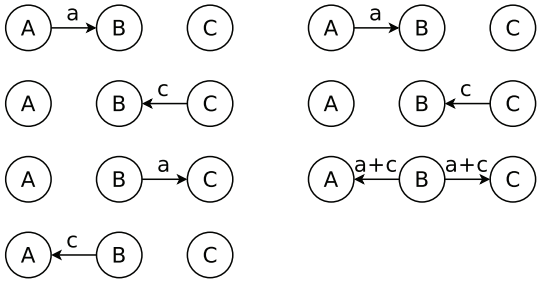


Throughput Dropoff

- **Only every third node can transmit, or you get the hidden terminal problem**
- **In TCP, data and ack packets cause the hidden terminal problem**



Bidirectional Network Coding (COPE, Katti et al.)



Coding

- **Reed-Solomon codes for burst errors on small data units**
- **LT codes for data delivery**
- **MORE for wireless communication**
- **General theme: being robust to individual losses through mixing data and redundancy**
- **Layer 1 (Reed-Solomon), Layer 2.5 (MORE, COPE), Layer 7 (LT)**