Lecture 3: Controlling Errors

CS 3035/GZ01: Networked Systems
Kyle Jamieson

Department of Computer Science
University College London
Recall (Lecture 1): Bit errors on links

• Links in a network go through hostile environments
  – Both wired, and wireless:

  – Consequently, errors will occur on links
  – Today: How can we control (i.e., detect and correct) these errors?

• There is a limited capacity available on any link
  – Tradeoff between link utilization and amount of error control
1. Error control codes
   – How many errors can we handle?
   – Measuring a code’s overhead: Code rate
2. Practical error control codes
3. The Internet checksum
The techniques we’ll discuss today are **pervasive** throughout the internetworking stack.

Based on theory, but broadly applicable in practice, in other areas:
- Hard disk drives
- Magnetic tape media
- Optical media (CD, DVD, & c.)
- Satellite, mobile communications
- Deep space, submarine communications

In Networked Systems, we cover just the basic concepts.
Error control in the Internet stack

- **Transport layer (L4)**
  - Internet Checksum (IC) over TCP or UDP header and data

- **Network layer (L3)**
  - Internet Checksum (IC) over IP header only

- **Link layer (L2)**
  - Cyclic Redundancy Check (CRC)

- **Physical layer (PHY)**
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
Error control: the fundamental problem

- Sender transmits a message to receiver through the network.
- Every string of bits is a possible legitimate message.
  - Hence any *errors* (changes) to the bits the sender transmits result in *equally-legitimate* messages.
- Therefore, errors happen, but without error control, receiver wouldn’t know about them!
Error control coding

• **Key idea: Reduce the set of legitimate messages**
  – Not every string of bits is an “allowed” message
  – Receipt of a disallowed string of bits means that the message was garbled in transit over the network

• We call an allowable message of $n$ bits a **codeword**
  – Not all $n$-bit messages are codewords!
  – The remaining $n$-bit strings are “space” between codewords

• We will use these ideas to both **detect** and **correct** errors in transmitted messages
Encoding and decoding

- **Problem:** Not every string of bits is “allowed”
  - But we want to be able to send any message!
  - How can we send a “disallowed” message?

- **Answer:** Sender-receiver protocol
  - The sender must **encode** its messages \(\rightarrow\) codewords
  - The receiver then **decodes** received bits \(\rightarrow\) messages

- We’ll look at simple codes today, but the relationship between messages and codewords isn’t always obvious!
A simple error-detecting code (1/4)

- Let’s start simple: suppose messages are one bit long.

- Let’s take each bit we want to send and **repeat** it once.
  - This is called a **two-repetition code**.

Sender:

- 0 → 00
- 0 → 01 (not allowed)
- 0 → 10 (not allowed)
- 1 → 11

Networked Systems 3035/GZ01
A simple error-detecting code (2/4)

- What happens at the receiver?
  - If the network makes no errors, the receiver removes the repetition to **correctly decode** the received codeword.

```
Sender: 0 → 00 → 00 → 0
         1 → 11 → 11 → 1

Network: 00 00
         11 11

Receiver: 0 1
```
A simple error-detecting code (3/4)

• If the network makes **one bit error**, receiver sees a non-codeword message and **detects an error**
  – Can the receiver **correct** the error?
  – **No!** The **other** codeword could have been sent as well
    • The receiver has no way of telling which codeword was sent in this case, so cannot correct the error

```
Sender: 0 ➜ 00 ➜ 00 ➜ 0
         01 ➜ 00 ➜ 01
         10 ➜ 00 ➜ 10

Network: 00 ➜ 00 ➜ 0
           01 ➜ 01
           10 ➜ 10

Receiver: 0 ➜ 0
           1 ➜ 1

Error detected
Error detected
```
A simple error-detecting code (4/4)

- Can the receiver detect the presence of **two bit errors**?
  - **No**: It has no way of telling which codeword was sent!
  - **What happened?** The network flipped so many bits that the message “jumped over” the space between codewords

- Let’s try to generalize this reasoning to **any** error-detecting code...

![Diagram showing sender, network, and receiver with codewords 0 and 1, and the space between codewords highlighted.]
**Hamming distance**

- How do we measure space between codewords?
  - Idea: Measure the number of bit flips to change one codeword into another
  - **Hamming distance** between two messages $m_1, m_2$: The number of bit flips needed to change $m_1$ into $m_2$

- **Example:** Two bit flips are needed to change codeword 00 to codeword 11, so they are Hamming distance of two apart
How many bit errors can we detect?

• Suppose the \textbf{minimum Hamming distance} between any pair of codewords is $d_{\text{min}}$
  – Some pairs may be separated by different Hamming distances than others

• Then, we can \textbf{detect} $\leq d_{\text{min}} - 1$ bit flips
  – This many are guaranteed to land in the space between codewords, as we just saw
  – Receiver will flag message as “\textbf{Error detected}”

\[
d_{\text{min}} = 3
\]
Decoding error detecting codes

- **Decoding** is a two-step process:

1. The receiver maps received bits $\rightarrow$ codewords
   - **Decoding rule**: Consider all codewords, choose the one that exactly matches received bits, or “error detected” if none match

2. The receiver maps codewords $\rightarrow$ source bits and “error detected”
A simple error-correcting code

• Let’s look at a three-repetition code. In the case of no errors it works like the two-repetition code we saw before.
A simple error-correcting code

- The receiver can fix one bit error
  - Chooses the closest codeword (in Hamming distance) to received bits
  - This amounts to drawing a decision boundary halfway between codewords and choosing closest

Sender:  
0 → 000  
  001  
  010  
  100  
  011  
  101  
  110  

Network:
000  
  011  
  010  
  100  
  011  
  101  
  110  

Receiver:  
0 → 000  
  011  
  010  
  100  
  011  
  101  
  110  

1 → 111  
  111  
  110  
  111  
  101  
  100  
  010  

Fix error

Decision boundary
Decoding error correcting codes

- **Decoding** is a two-step process:

  1. The receiver maps received bits → codewords
     - **Decoding rule**: Consider all codewords, select one with the **minimum Hamming distance** to the received bits

  2. The receiver maps codewords → source bits
How many bit errors can we correct?

- Suppose there is $\geq d_{\text{min}}$ Hamming distance between any codeword pair
- Then, we can correct $\leq \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor$ bit flips
  - This many bit flips can’t move received bits closer to another codeword, across the decision boundary:

\[ d_{\text{min}} = 5 \]

Decision boundary
Code rate

- Suppose codewords are of length $n$, messages length $k$

- There is overhead involved in controlling errors: codewords are as long or longer than the messages they describe

- The code rate $R = k/n$ measures this overhead

- So, we have a tradeoff:
  - High-rate codes ($R$ approaching one) correct fewer errors, but add less overhead
  - Low-rate codes ($R$ close to zero) correct more errors, but add more overhead
Today

1. Error detecting and correcting codes

2. Practical error control codes
   - Parity, and 2-D parity
   - Block codes: Hamming (7, 4) code
   - Polynomial codes: Cyclic redundancy check

3. The Internet checksum
Parity bit

- Fundamental building block for most other codes

- Given a message of \( k \) data bits \( D_1, D_2, \ldots, D_k \), append a parity bit \( P \) to make a codeword of length \( n = k + 1 \)
  - Pick the parity bit so that total number of 1’s is even
  - The parity bit \( P \) is the exclusive-or of the data bits:
    - \( P = D_1 \oplus D_2 \oplus \cdots \oplus D_k \)

\[
\begin{array}{c}
\text{k data bits} \\
011100
\end{array} \quad \begin{array}{c}
\text{parity bit} \\
1
\end{array}
\]
Checking the parity bit

• **At the receiver:** is there an even number of 1’s in the received message?
  – If so, received message is a codeword
  – If not, isn’t a codeword, and error detected
    • But receiver doesn’t know where, so *can’t correct*

• Consider minimum distance between codewords $d_{\text{min}}$:
  – Change one data bit $\rightarrow$ change parity bit, so $d_{\text{min}} = 2$
  – Therefore, a parity bit detects at most one bit error, corrects zero

• Can we detect and correct more errors, in general?
Two-dimensional parity

• Let’s see what happens if we generalize the parity check to two dimensions

• Break up data into multiple rows
  – Start with normal parity within each row \( (p_i) \)
  – Do the same across rows \( (q_i) \)
  – Add a parity bit \( r \) covering row parities

• This code has rate 16/25
Two-dimensional parity: Properties

• What is $d_{\text{min}}$ for this code?
  – Change any one data bit, three parity bits change
  – Change any two data bits, **at least** two parity bits change
  – Change any three data bits, **at least** three parity bits change

• Therefore, $d_{\text{min}} = 4$, so
  – 2-D parity can detect one-, two-, and three-bit errors
  – 2-D parity can correct single-bit errors (parity bits pick out the row and column)

• 2-D parity detects **most** four-bit errors
1. Error detecting and correcting codes

2. Practical error control codes
   - Parity, and 2-D parity
   - Block codes: Hamming (7, 4) code
   - Polynomial codes: Cyclic redundancy check

3. The Internet checksum
Block codes

• Let’s **fully generalize the parity bit** for even more error detecting/correcting power

• Split message into $k$-bit blocks, and add $n-k$ parity bits to the end of each block:
  – This is called an $(n, k)$ **block code**

![Diagram of a codeword: $n$ bits made up of $k$ data bits and $n-k$ parity bits.](chart)
Correcting errors with parity bits

• Can we build a high-rate code to *correct* 1-bit errors?
  – Repetition coding is inefficient: needs **three** parity bits per data bit: \((R = \frac{1}{4})\)
  – **Possible** with 3 parity per 4 data bits \((R = 4/7 \approx 0.57)\)

• What if we **repeat the parity bit 3×**?
  – \(P = D_1 \oplus D_2 \oplus D_3 \oplus D_4; R = 4/7\)
  – Change one data bit, all parity bits flip. So \(d_{\text{min}} = 4?\)
    • **No!** Change another data bit, all parity bits flip back to their original values! So \(d_{\text{min}} = 2\)
  – No error correcting, one bit error detecting capability
  – **What happened?** Parity checks either all failed or all succeeded, giving no additional information
Hamming (7, 4) code

- Have each data bit in a different subset of parity bits
  - So, an error in a certain data bit shows up as a certain set of inconsistent parity checks
  - There are three parity bit subsets of size two, and one of size three

\[ k = 4 \text{ bits} \quad n - k = 3 \text{ bits} \]

\[ P_1 = D_1 \oplus D_3 \oplus D_4 \]

\[ P_2 = D_1 \oplus D_2 \oplus D_3 \]

\[ P_3 = D_2 \oplus D_3 \oplus D_4 \]
Hamming (7, 4) code

• How many errors can this code detect and correct?
  – Depends on $d_{\text{min}}$

• Change one data bit, either:
  – Two $P_i$ change, or
  – Three $P_i$ change

• Change two data bits, either:
  – Two $P_i$ change, or
  – One $P_i$ changes

• Therefore, $d_{\text{min}} = 3$
  – So can detect two bit errors, correct one bit error

Networked Systems 3035/GZ01
Hamming (7, 4) code in practical use

- Suppose the network can corrupt at most one bit
  
- Can “read off” the corrupt bit from which parity checks fail:
  - Parity checks $P_1$ and $P_2$ fail $\Rightarrow$ Error in $D_1$
  - Parity checks $P_2$ and $P_3$ fail $\Rightarrow$ Error in $D_2$
  - Parity checks $P_1$, $P_2$, and $P_3$ fail $\Rightarrow$ Error in $D_3$
  - Parity checks $P_1$ and $P_3$ fail $\Rightarrow$ Error in $D_4$

- One parity check fails $\Rightarrow$ data is okay, but parity bit was corrupted

- No parity checks fail $\Rightarrow$ data and parity okay
Today

1. Error detecting and correcting codes

2. Practical error control codes
   – Parity, and 2-D parity
   – Block codes: Hamming (7, 4) code
   – Polynomial codes: Cyclic redundancy check

3. The Internet checksum
Error detection at the link layer: CRC

- **Problem:** Errors overcome the PHY’s error-control capability
  - Need **error detection** with a low probability of missing an error

- **Link layer (L2)**
  - Cyclic Redundancy Check (CRC)
  - L2 CRC improves performance
    - Quickly filters out corrupted frames

- **Physical layer (PHY)**
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
Cyclic redundancy check (CRC)

- Most popular method **error detecting code** at L2
  - Found in Ethernet, WiFi, token ring, many many others

- Often implemented in hardware at the link layer

- Represent $k$-bit messages as degree $k - 1$ **polynomials**
  - Each coefficient in the polynomial is either zero or one, *e.g.*:

\[
M(x) = 1x^5 + 0x^4 + 1x^3 + 1x^2 + 1x + 0
\]

$k = 6$ bits of message

\[
\begin{array}{ccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]
Modulo-2 arithmetic

• Addition and subtraction are both exclusive-or, without carry or borrow

Multiplication example:

\[
\begin{array}{r}
  1101 \\
  \underline{\times 110} \\
  0000 \\
  11010 \\
  110100 \\
  \underline{101110}
\end{array}
\]

Division example:

\[
\begin{array}{r}
  1101 \\
  \underline{110} \overline{101110} \\
  110 \\
  110 \\
  \underline{111} \\
  110 \\
  \underline{011} \\
  000 \\
  \underline{110} \\
  110
\end{array}
\]
**CRC at the sender**

- **M(x)** is our **message** of length **k**
  - *e.g.:* $M(x) = x^5 + x^3 + x^2 + x$ ($k = 6$)  
    
  
    \[
    \begin{array}{c}
    1 \ 0 \ 1 \ 1 \ 1 \ 0
    \end{array}
    \]

- Sender and receiver agree on a **generator** polynomial $G(x)$ of degree $g - 1$ (*i.e.*, $g$ bits)
  - *e.g.:* $G(x) = x^3 + 1$ ($g = 4$)  
    
  
    \[
    \begin{array}{c}
    1 \ 0 \ 0 \ 1
    \end{array}
    \]

1. Calculate **padded message** $T(x) = M(x) \cdot x^{g-1}$
   - *i.e.*, right-pad with $g - 1$ zeroes
   - *e.g.:* $T(x) = M(x) \cdot x^3 = x^8 + x^6 + x^5 + x^4$  
    
    \[
    \begin{array}{c}
    1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \\
    \end{array}
    \]
CRC at the sender

2. Divide padded message $T(x)$ by generator $G(x)$
   - The remainder $R(x)$ is the CRC:

\[
\begin{array}{cccccccc}
  & & & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

$R(x) = x + 1$
CRC at the sender

3. The sender transmits codeword $C(x) = T(x) + R(x)$
   – i.e., the sender transmits the original message with the CRC bits appended to the end
   – Continuing our example, $C(x) = x^8 + x^6 + x^5 + x^4 + x + 1$

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]
Properties of CRC codewords

• Remember: \textbf{Remainder} \left[ \frac{T(x)}{G(x)} \right] = R(x)

• What happens when we divide \( C(x) / G(x) \)?

• Codeword is \( T(x) + R(x) \) so the remainder is
  – \textbf{Remainder} \left[ \frac{T(x)}{G(x)} \right] = R(x), \text{ plus}
  – \textbf{Remainder} \left[ \frac{R(x)}{G(x)} \right] = R(x)

  – Recall, addition is exclusive-or operation, so:
    • \textbf{Remainder} \left[ \frac{C(x)}{G(x)} \right] = R(x) + R(x) = 0
Detecting errors at the receiver

- Divide received message $C'(x)$ by generator $G(x)$
  - If no errors occur, remainder will be zero

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & | & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & | & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & & & & & \\
1 & 0 & 0 & 1 & & & & & & \\
0 & 1 & 0 & 1 & & & & & & \\
0 & 0 & 0 & 0 & & & & & & \\
1 & 0 & 1 & 0 & 1 & & & & & \\
1 & 0 & 1 & 0 & 1 & & & & & \\
0 & 1 & 1 & 0 & & & & & & \\
0 & 0 & 0 & 0 & & & & & & \\
1 & 1 & 0 & 1 & & & & & & \\
1 & 1 & 0 & 1 & & & & & & \\
1 & 0 & 0 & 1 & & & & & & \\
1 & 0 & 0 & 1 & & & & & & \\
0 & 0 & 0 & 0 & \rightarrow \text{no error detected}
\end{array}
\]
Detecting errors at the receiver

• Divide received message $C'(x)$ by generator $G(x)$
  – If errors occur, remainder may be non-zero

\[
\begin{array}{cccccc}
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & \end{array}
\]

\[1001 \quad \overline{1011111011} \quad \frac{1001}{1001} \quad \frac{0101}{0000} \quad \frac{101}{1001} \quad \frac{010}{0000} \quad \frac{1001}{1001} \quad \frac{0100}{0000} \quad \frac{1001}{1001} \quad \frac{0001}{0001} \quad \rightarrow \text{error detected}\]
Detecting errors at the receiver

• Divide received message $C'(x)$ by generator $G(x)$
  – If errors occur, remainder may be non-zero

How many errors can the CRC detect?

How do we choose generator $G(x)$?

0 0 0 0 → undetected error!
Detecting errors with the CRC

- The error polynomial \( E(x) = C(x) + C'(x) \) is the difference between the transmitted and received codeword
  - \( E(x) \) tells us which bits the channel flipped

- We can write the received message \( C'(x) \) in terms of \( C(x) \) and \( E(x) \): \( C'(x) = C(x) + E(x) \), so:
  - \( \text{Remainder} \left[ \frac{C'(x)}{G(x)} \right] = \text{Remainder} \left[ \frac{E(x)}{G(x)} \right] \)

- When does an error go undetected?
  1. When \( \text{Remainder} \left[ \frac{E(x)}{G(x)} \right] = 0 \), or
  2. When \( E(x) = G(x) \cdot Z(x) \) for some polynomial \( Z(x) \)
Detecting single-bit errors with the CRC

• Suppose a single-bit error in bit-position \( i: E(x) = x^i \)
  
  – Choose \( G(x) \) with \( \geq 2 \) non-zero terms: \( x^{g-1} \) and 1
  
  – Remainder \( [ x^i / (x^{g-1} + \cdots + 1) ] \neq 0, \text{e.g.:} \)

\[
\begin{array}{c|c}
1 & 001000 \\
\hline
1001 & 1001 \\
& 1
\end{array}
\]

• Therefore a **CRC with this choice of** \( G(x) \) **always detects single-bit errors** in the received message
Implementing the CRC in hardware

• Implemented in hardware at L2 using a shift register

• Shift register performs division
  – Think of time as discrete clock “ticks”
  – Every clock tick the register copies its input to its storage and outputs what it stores
  – $g-1$ registers hold partial sum of each division step
  – Exclusive-or gates $\oplus$ subtract generator polynomial

\[ T(x) = x^3 + x^2 + x^1 + x^0 \]
Implementing the CRC in hardware

- Register begins zero-initialized
- Shift $T(x)$ in, one bit at a time
  - Each shift corresponds to one long-division step
- A 1 is in the $x^3$ register whenever 1 is in the quotient
  - The xor gate then “subtracts” $G(x)$ (xor gates placed in same pattern as 1’s in $G(x)$)
Error detecting properties of the CRC

• The CRC will detect:
  ✔ All single-bit errors
    • Provided $G(x)$ has two non-zero terms

  – All burst errors of length $\leq g - 1$
    • Provided $G(x)$ begins with $x^{g-1}$ and ends with 1
    • Similar argument to previous property

  – All double-bit errors
    • With conditions on the frame length and choice of $G(x)$

  – Any odd number of errors
    • Provided $G(x)$ contains an even number of non-zero coefficients
Common generators

- CRC-8: $x^8 + x^2 + x + 1$
- CRC-10: $x^{10} + x^9 + x^5 + x^4 + x + 1$
- CRC-16: $x^{16} + x^{15} + x^2 + 1$
- CRC-CCITT: $x^{16} + x^{12} + x^5 + 1$

- These generator polynomials meet certain properties:
  - At least two non-zero terms
  - Even number of non-zero coefficients
Today

1. Error detecting and correcting codes
2. Practical error control codes
3. The Internet checksum
L3/L4 error detection: the *Internet Checksum*

- **Transport layer (L4)**
  - Internet Checksum (IC) over TCP or UDP header and data
  - End to end (E2E) checksum

- **Network layer (L3)**
  - Internet Checksum (IC) over IP header only

- **Link layer (L2)**
  - Cyclic Redundancy Check (CRC)
  - L2 CRC improves performance

- **Physical layer (PHY)**
  - Forward Error Correction (FEC) or Error Control Coding (ECC)
Why another error detecting code?

- Most errors are picked up at lower layers; this is the last layer that can detect errors before the application itself
  - L2 corrects most errors with the CRC

- Implemented in software: simple and fast

- The only checksum that works **end-to-end** at the transport layer (L4)
  - Why is that important?
Ones’ complement arithmetic

- To represent $x$ in ones’ complement:
  - If $x$ is positive, write $x$ in binary and prepend a zero
    - E.g. $5_{10} = 0101$
  - If $x$ is negative, write $|x|$ in binary, negate it (complementing each bit), and prepend a one
    - E.g. $-5_{10} = 1010$

- Notice that there are two ways to represent zero

- Addition uses carry and “end-around” carry: $-5 - 3$, e.g.:
  \[
  \begin{array}{c}
  1110: \quad -1_{10} \\
  1100: \quad -3_{10} \\
  \hline
  1011: \quad -4_{10} \times
  \end{array}
  \]
Internet checksum

• **At the sender:**
  1. Break packet into 16-bit words (unsigned short or “ushort” data type)
  2. Sum the words together using ones’ complement addition
  3. Negate the result of the sum, and include in the relevant (IP or TCP) header field

• **Receiver** performs same calculation on received data
  – If result matches, receiver declares that there are no errors

Internet checksum C code:

```c
ushort cksum(ushort *buf, int count) {
    ulong sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* end-around carry */
            sum = sum & 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0x0000FFFF);
}
```
Internet checksum: Properties

• Space efficient, using one word (16 bits) for any message length

• Recomputing checksum $C$ after changing a single word $m \rightarrow m'$
  – Negate checksum, subtract $m$, add $m'$, negate result
  – **New checksum $C'$** = $\sim(\sim C + \sim m + m')$

• But, the Internet checksum has weaker error detection properties than the CRC:
  – Checksum is the negation of sum of data, so any single-bit error in message or data is detectable
  – No guarantees beyond that an undetectable double-bit error:

<table>
<thead>
<tr>
<th>Word #1</th>
<th>0 ... 0000 0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word #2</td>
<td>0 ... 0000 0100</td>
</tr>
<tr>
<td>#1 + #2</td>
<td>0 ... 0000 1001</td>
</tr>
<tr>
<td>Checksum</td>
<td>1 ... 1111 0110</td>
</tr>
</tbody>
</table>
Medium Access Control

Pre-Reading: P & D Section 2.6 (5/e, 4/e)

Pre-Reading: Ethernet Paper (PDF available online)

NEXT TIME