Overview

• Goal: transmit correct information
• Problem: bits can get corrupted
  – Electrical interference, thermal noise
• Problem: packets can be lost

• Solution
  – Detect errors
  – Recover from errors
    • Correct errors
    • Retransmission
Outline

- Revisit error detection
  - Reliable Transmission

Naïve approach

- Send a message twice
- Compare two copies at the receiver
  - If different, some errors exist

- How many bits of error can you detect?

- What is the overhead?
Error Detection

- Problem: detect bit errors in packets (frames)
- Solution: add extra bits to each packet
- Goals:
  - Reduce overhead, i.e., reduce the number of redundancy bits
  - Increase the number and the type of bit error patterns that can be detected
- Examples:
  - Two-dimensional parity
  - Checksum
  - Cyclic Redundancy Check (CRC)
  - Hamming Codes

Parity

- Even parity
  - Add a parity bit to 7 bits of data to make an even number of 1’s

\[
\begin{align*}
0110100 & \quad 1 \\
1011010 & \quad 0
\end{align*}
\]

- How many bits of error can be detected by a parity bit?
- What's the overhead?
Two-dimensional Parity

• Add one extra bit to a 7-bit code such that the number of 1’s in the resulting 8 bits is even (for even parity, and odd for odd parity)
• Add a parity byte for the packet
• Example: five 7-bit character packet, even parity

<table>
<thead>
<tr>
<th>Packet</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011010</td>
<td>0</td>
</tr>
<tr>
<td>0010110</td>
<td>1</td>
</tr>
<tr>
<td>1110101</td>
<td>1</td>
</tr>
<tr>
<td>1001011</td>
<td>0</td>
</tr>
<tr>
<td>1000110</td>
<td>1</td>
</tr>
</tbody>
</table>

How Many Errors Can you Detect?

• All 1-bit errors
• Example:

<table>
<thead>
<tr>
<th>Packet</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110100</td>
<td>1</td>
</tr>
<tr>
<td>1011010</td>
<td>0</td>
</tr>
<tr>
<td>0000110</td>
<td>1</td>
</tr>
<tr>
<td>1110101</td>
<td>1</td>
</tr>
<tr>
<td>1001011</td>
<td>0</td>
</tr>
<tr>
<td>1000110</td>
<td>1</td>
</tr>
</tbody>
</table>
How Many Errors Can you Detect?

• All 2-bit errors
  • Example:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0110100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011010</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000111</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101011</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001011</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000110</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

odd number of 1's on columns

error bits

How Many Errors Can you Detect?

• All 3-bit errors
  • Example:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0110100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011010</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000111</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101011</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001011</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000110</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

odd number of 1's on column

error bits
How Many Errors Can you Detect?

- Most 4-bit errors
- Example of 4-bit error that is not detected:

```
0110100 1
1011010 0
0000111 1
1100100 1
1001011 0
1000110 1
```

How many errors can you correct?

Checksum

- Sender: add all words of a packet and append the result (checksum) to the packet
- Receiver: add all words of a received packet and compare the result with the checksum
- Example: Internet checksum
  - Use 1’s complement addition
1’s Complement

- Negative number \(-x\) is \(x\) with all bits inverted
- When two numbers are added, the carry-on is added to the result
- Example: \(-15 + 16\); assume 8-bit representation

\[
\begin{align*}
15 &= 00001111 \\
\Rightarrow -15 &= 11110000 \\
16 &= 00010000 \\
\Rightarrow -15 + 16 &= 00000001
\end{align*}
\]

Internet Checksum Implementation

```c
u_short csum(u_short *buf, int count) {
    register u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```
Properties

- How many bits of error can Internet checksum detect?
- What’s the overhead?
- Why use this algorithm?
  - Link layer typically has stronger error detection
  - Most Internet protocol processing in the early days (70’s 80’s) was done in software with slow CPUs, argued for a simple algorithm
  - Seems to be OK in practice
- What about the end-to-end argument?

Example of checksum calculation

- If data is
  
  1001 1101 0010 1101 1100 0011 1101 0101

- Convert to 16-bit words, then add, carry, and invert

  1001 1101 0010 1101
  1100 0011 1101 0101
  0110 0001 0000 0010  Sum
  0110 0001 0000 0011  Carry
  0110 0001 0000 0011  Final sum

  1001 1110 1111 1100  Internet checksum
Overview

- Revisit error detection
  - Reliable transmission

Retransmission

- Problem: obtain correct information once errors are detected
- Retransmission is one popular approach
- Algorithmic challenges
  - Achieve high link utilization, and low overhead
Reliable Transfer

- Retransmit missing packets
  - Numbering of packets and ACKs
- Do this efficiently
  - Keep transmitting whenever possible
  - Detect missing ACKs and retransmit quickly
- Two schemes
  - Stop & Wait
  - Sliding Window
    - Go-back-n and Selective Repeat variants

Stop & Wait

- Send; wait for acknowledgement (ACK); repeat
- If timeout, retransmit

Inefficient if TRANS << RTT
Stop & Wait

- TRAN
- DATA
- Receiver
- Sender
- Timeout
- Time

Is a Sequence Number Needed?

- Need a 1 bit sequence number (i.e. alternate between 0 and 1) to distinguish duplicate frames
Problem with Stop-and-Go

- Lots of time wasted in waiting for acknowledgements

- What if you have a 10Gbps link and a delay of 10ms?
  - Need 100Mbit to fill the pipe with data

- If packet size is 1500B (like Ethernet), because you can only send one packet per RTT
  - Throughput = 1500*8bit/(2*10ms) = 600Kbps!
  - A utilization of 0.006%

Sliding Window

- window = set of adjacent sequence numbers
- The size of the set is the window size (WS)
  - Assume it is n

- Let A be the last ack’d packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
  - Sender window size (SWS)

- Sender can send packets in its window

- Let B be the last received packet without gap by receiver, then window of receiver = {B+1, …, B+n}
  - Receiver window size (RWS)

- Receiver can accept out of sequence packets, if in window
Basic Timeout and Acknowledgement

• Every packet k transmitted is associated with a timeout
• If by timeout(k), the ack for k has not yet been received, the sender retransmits k

• Basic acknowledgement scheme
  – Receiver sends ack for packet k when all packets with sequence numbers ≤ k have been received
  – An ack k means every packet up to k has been received

  Suppose packets A, B, C, D have been received, but receiver is still waiting for A. No ack is sent when receiving B, C, D. But as soon as A arrives, an ack for D is sent by the receiver, and the receiver window slides...
Example with Errors

Window size = 3 packets

Sender

Timeout Packet 5

5 6 7

1 2 3 4 5

Receiver

Time

Efficiency

SWS = 9, i.e. 9 packets in one RTT instead of 1

→ Can be fully efficient as long as WS is large enough
Observations

• With sliding windows, it is possible to fully utilize a link, provided the window size is large enough. Throughput is \( \sim (n/\text{RTT}) \)
  – Stop & Wait is like \( n = 1 \).
• Sender has to buffer all unacknowledged packets, because they may require retransmission
• Receiver may be able to accept out-of-order packets, but only up to its buffer limits

Setting Timers

• The sender needs to set retransmission timers in order to know when to retransmit a packet that may have been lost
• How long to set the timer for?
  – Too short: may retransmit before data or ACK has arrived, creating duplicates
  – Too long: if a packet is lost, will take a long time to recover (inefficient)
Timing Illustration

- Timeout too long → inefficiency
- Timeout too short → duplicate packets

Adaptive Timers

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
- For link-layer networks (LANs), this value is essentially known
- For multi-hop WANS, rarely known
- Must work in both environments, so protocol should adapt to the path behavior
- E.g. TCP timeouts are adaptive, will discuss later in the course