Lecture 22: Reliable transport

Prof. Alan Mislove (amislove@ccs.neu.edu)

Slides used with permissions from Edward W. Knightly, T. S. Eugene Ng, Ion Stoica, Hui Zhang
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

  0110100

  1011010

• How many bits of error can be detected by a parity bit?
• What’s the overhead?
Parity

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

  0110100
  \hspace{1cm} 1
  \hspace{1cm} 1011010

• How many bits of error can be detected by a parity bit?
• What’s the overhead?
Parity

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

\[ \begin{array}{c}
0110100 \\
1011010
\end{array} \quad \begin{array}{c}
1 \\
0
\end{array} \]

• 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

0110100
1011010
0010110
1110101
1001011
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

0110100
1011010
0010110
1110101
1001011

1
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

0110100 1
1011010 0
0010110
1110101
1001011
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></td>
</tr>
<tr>
<td>1001011</td>
<td></td>
</tr>
</tbody>
</table>
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
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

```
0110100  1
1011010  0
0010110  1
1110101  1
1001011  0
```
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

```
0110100
1011010
0010110
1110101
1001011
```

1
0
1
1
0
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
How Many Errors Can you Detect?

• All 1-bit errors
• Example:

```
  0110100  1
  1011010  0
  0000110  1
  1110101  1
  1001011  0
  1000110  1
```

error bit
How Many Errors Can you Detect?

• All 1-bit errors

Example:

```
0110100
1011010
0011001
1110101
1001011
```

- error bit
- odd number of 1's
How Many Errors Can you Detect?

• All 2-bit errors
• Example:

```
0110100
1011010
0000111
1110101
1001011
1000110

error bits
odd number of 1's on columns
```
How Many Errors Can you Detect?

- All 3-bit errors
- Example:

```
0110100
1011010
0000111
1100101
1001011
1000110
```

- odd number of 1’s on column
How Many Errors Can you Detect?

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

```
0110100
1011010
 00
 011
 1
```

error bits

```
0000111
```

```
1100100
```

```
1001011
```

```
1000110
```

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

\[
15 = 00001111 \rightarrow -15 = 11110000 \\
+ \\
16 = 00010000
\]
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

\[
15 = 00001111 \rightarrow -15 = 11110000 \\
+ \\
16 = 00010000 \\
\overline{1} 00000000
\]
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

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

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

\[
\begin{align*}
00000000 &+ 1 \quad 00000000 \\
\hline
00000001 &+ 1 \\
\hline
-15 + 16 &= 1
\end{align*}
\]
Internet Checksum Implementation

```c
u_short cksum(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
Properties

• How many bits of error can Internet checksum detect?
Properties

- How many bits of error can Internet checksum detect?
- What’s the overhead?
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
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

  \[
  \begin{array}{cccccccc}
  1001 & 1101 & 0010 & 1101 \\
  1100 & 0011 & 1101 & 0101 \\
  \hline
  \text{Sum} & 10110 & 0001 & 0000 & 0010 \\
  \text{Carry} & 0110 & 0001 & 0000 & 0011 \\
  \text{Final sum} & 1001 & 1110 & 1111 & 1100 \\
  \end{array}
  \]

  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 \( \text{TRANS} \ll \text{RTT} \)

\[ \text{RTT} \quad \text{Round-Trip-Time} \]
Is a Sequence Number Needed?

timeout

Frame

ACK

Frame

ACK

timeout

Frame

ACK

Frame

ACK
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
Problem with Stop-and-Go

• Lots of time wasted in waiting for acknowledgements
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 = $1500 \times 8\text{bit}/(2 \times 10\text{ms}) = 600\text{Kbps}$!
  – A utilization of 0.006%
Sliding Window

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

- Let \( A \) be the last acknowledged packet of the sender without gap; then window of sender = \( \{A+1, A+2, \ldots, A+n\} \)
  - Sender window size (SWS)

- Sender can send packets in its window

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

- Receiver can accept out of sequence packets, if in window
Example

SWS = 9

Time
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 packet A B C D e 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

Timeout
Packet 5

Sender

Receiver
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 \( \sim \frac{n}{\text{RTT}} \)
  – 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.