Announcements

Tutorials start in weeks 3/4

Allocations posted in webCT “Labs – general” section

Labs start in weeks 3/4

- Bring a signed “Declaration of Originality” (from Course Outline – hardcopy from 1st lecture, or electronic copy from webCT)
- Description of the first lab exercise will be on the course web page on Monday.
  - First lab is essentially Q38 on p. 346 of Tanenbaum about CRCs
  - Error checking & CRCs covered in next lectures (some are unfortunately after the first labs)
Course outline

• Introduction to networks and protocols
• Analog domain: Apps & transmission media
• Links
• Medium access control
• Networks
• End-to-end transport
• Applications
Section outline

Links:
• Framing
• Error coding
  o For error detection
  o For error correction
• Error correction through retransmission

Tanenbaum sections
• 3.1.2
• 3.1.3
  o 3.2.2
  o 3.2.1
• 3.3 and 3.4
We’ll discuss causes of errors in the context of the Physical Layer.

- Significance of errors
- Types of errors
- Error control: Protection, Detection, Correction
- Error codes:
  - Parity
  - Error correcting codes
  - Checksums
  - Cyclic Redundancy Codes

Image from A. K. Dewdney: *The Turing Omnibus*, p. 305
Significance of errors

If errors occur in ...

... payload information:
  o Executable program could crash if it contains errors – need high integrity.
  o May deliver misinformation
    e.g. “Transfer $1,000” → “Transfer $9,000”†
  o Samples of streaming media (e.g. video pixels) may be relatively insensitive to bit errors.
    Compressed streaming media is more sensitive to error.

... control information:
  o Errors may cause protocols to operate incorrectly,
    e.g. errors in sequence numbers may result in information being delivered in the wrong sequence.

† In this section we only consider “errors” that have predictable properties, e.g. each bit has equal probability of being in error. When the cause of “error” is a malicious attacker, then the errors may be targeted to try to circumvent protective measures. In such cases, cryptographically strong integrity checks (e.g. secure hash functions) are needed.
Locating error control

Error control is often repeated in multiple layers of the protocol stack.

2 common layers:

- **Link layer**
  - Low cost to check here (must handle every bit in any case)
  - Often implemented in hardware $\Rightarrow$ sophisticated error checks (e.g. CRC-32)

- **Transport layer**
  - “End-to-end arguments” indicate that checking is needed here, even if it is also provided at other layers as a performance enhancement.
  - Often implemented in software $\Rightarrow$ simpler error checks (e.g. Internet checksum)

Other layers: They have to ensure that their own headers are protected
- e.g. IPv4 header checksum. Corruption of address could result in delivery to wrong destination.
Types of errors

**Inversion**: Bit values are flipped: $0 \rightarrow 1$, $1 \rightarrow 0$.

Process that causes errors may:
- Directly invert bits
- Polarise bits; only bits with opposite polarity are inverted.

Knowing location enables correction.

**Erasure**: Set of contiguous bits are lost (or reset to a known value)

- Location is known; value is not known

**Insertion/Deletion**: Set of bits are injected or removed from the midst of the string.

Original: 010101000110100101101101

Received: 01110000110111001101101

Received: 01010100--------01101101

Received: 01010100100101001101101

Deleted bits: 0110

Example

Original: 010101000110100101101101

Received: 01110000110111001101101

Received: 01010100--------01101101

Received: 01010100100101001101101

Deleted bits: 0110

Usually assume that this type of error can be dealt with by mechanisms for dealing with other types, e.g. bits shifted to new positions will, with $P=0.5$, have wrong value for those positions ⇒ detect with mechanisms for detecting inversion.
Bursty errors

Errors are often caused by physical phenomena persisting for an interval
⇒ May affect multiple bits transmitted during that interval
⇒ Errors often occur in bursts

Bursty errors:
✓ concentrate damage in certain packets
  e.g. if we’re damaged packets will be retransmitted, we’d prefer to have one packet badly damaged than many packets partially damaged.
× harder to detect
Error control

3 steps:
1. **Protection**: Add redundancy to the payload.
   - Transmission
   - Reception (probably – detect loss by using sequence numbers)
2. **Detection of errors**
   Generally based on checking redundancy added during the protection step.
3. **Correction of errors**
   May involve retransmission by the source, or use of redundancy transmitted with the payload.
Primitive error checking

Idea: Send the information multiple times
   e.g. "TELE3018" → "TELE3018TELE3018TELE3018"
If the received items:
   differ:
      e.g. "TELE3018TELE3013TELE3018"
      then an error is detected
      • Request retransmission, or
      • Correct error by assuming majority is correct.
   are the same: *probably* no error.
      e.g. "TELE3018TELE3018TELE3018"
      or "TELE3013TELE3013TELE3013"

Majority ⇒ must send at least 3 times.
Can we use less redundancy & so reduce the overhead?
Yes: Shannon’s Channel Coding Theorem
Error checking is probabilistic

False detection:
- Might detect an “error” when in fact the payload is error-free, and it is the redundancy that is in error.
- Transmission errors can occur in the redundancy, just as they can occur in the payload.
- Assuming that the error rate is the same in the payload and redundancy, making payload >> redundancy:
  - reduces chance of false detection
  - improves efficiency

False non-detection:
“probably no error”: no guarantees. e.g. when transmitted 3 times, it is possible that 2 transmissions were corrupted in the same way.
Generalisation of basic error checking

Essential feature is that valid information follows a known form:
- Source ensures that what it transmits follows the known form.
- Destination checks that the received information also has this form.
Known form is generally achieved by adding redundancy to the information.

Some content inherently contains redundancy, e.g. human languages:
"this phrase makes sense"
"ths phrs mks sns": Certain letters (e.g. vowels) convey little information
"thus phrase makes sense": Grammatical rules help error detection.

However, networks are designed to separate content from carrier (e.g. OSI presentation layer from reliable transport layer) & can’t rely on the content having a certain form
⇒ Artificially add redundancy of known form to ensure that it is present.

The improvement in integrity comes at the cost of lower effective throughput (for given channel capacity).
Codes

Codes replace one word (string of bits) with another.
E.g. "000"→"0001", "001"→"0010", "010"→"0100", ...

Terminology:
- Block\( k \) of \( k \) bits of data/payload
- Add \( n-k \) bits of redundancy to create an \( n \)-bit codeword.
- Code often denoted as (\( n,k \))
  E.g. in a "(7,4) code", each codeword has 7 bits, 4 of them carrying data
- Codewords can be either:
  - valid (redundancy has the required form), or
  - invalid (likely a valid codeword that has been errored)
Hamming weight and distance

**Hamming weight** of a codeword: Number of 1s in the codeword
- e.g. 01101001 has a weight of 4

**Hamming distance** between codewords: Number of bits where they differ
- = weight of result of exclusive-ORing codewords
  - e.g. 01010100 01101001 01101101
  - and 01010100 01101111 01101101
  - have a Hamming Distance of 2

**Hamming distance of a code**: $d = \min(\text{Hamming distance between any 2 codewords})$

Requisite Hamming distance between codewords:
- To detect $e$ errors: $d = e + 1$
- To correct $e$ errors: $d = 2e + 1$

(Some) Error detection & correction codes

Mathematical basis:

Exclusive-OR
↓
Parity
↓
bit-based  word-based  multi-dimensional

Addition  Division
↓  ↓
Checksums  CRCs

2-dimensional  Hamming

Message Digests
(See Security chapter)
can also be used to check integrity.

There exist many other forms of codes for error detection & correction
Outline
Parity checking

Add an extra bit whose value is such that the total number of 1s in the payload+extra_bit is either
- even: Even parity
- odd: Odd parity

(Generally prefer odd parity s.t. we get some signal variations when transmitting information, e.g. 0V=0, 1V=1 don’t transmit 0s indefinitely – lose clock and unsure whether circuit is broken.)

Properties:
- Can detect any single error
- Can detect any odd number of errors
- Can’t detect even number of errors
Parity checking examples

Protect 01010100 by adding
1 for even parity: 010101001
0 for odd parity: 010101000

Detecting errors (odd parity):
- 1 error:
  011101000 = even parity ⇒ error detected
- odd number of errors (5):
  101011000 = even parity ⇒ error detected
- even number (2) of errors:
  001101000 = odd parity ⇒ error undetected
### Receiver processing of parity

Either:
- **Calculate parity over data**
  - e.g. odd parity for \(01110100\) = 1
- or
- **data and parity**
  - e.g. parity for \(01110100\) = even

**Compare to parity received**
- e.g. received 0
  - \(1 \neq 0 \Rightarrow \text{error detected}\)

**expected value**
- even\(\neq\)odd \(\Rightarrow\) error detected

Receiver generally discards packets that fail the integrity check.
- Higher layer retransmission protocols recover from this loss
  - (in the same manner by which they recover from whole packets being lost inside the network)

† Indeed, of any error detecting code, also including checksums and CRCs
Word-based parity

Rather than doing bit-wise calculations (time consuming on computer with multiple bits per word), do word-wise calculations e.g. exclusive-OR one word with another

Original: 010101000110100101101101

⊕

01101001

⊕

01101001

⇒

even parity check = 01010000

even number of 1s in each column

for odd parity, take 1s-complement of parity word
Outline
Internet checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into checksum header field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless*

Used in IPv4 header & TCP/UDP transport layers.
Outline
Two-dimensional parity

Can correct single errors by localising them:
- In payload: mismatch in both row and column parity
- In parity: Error in parity bits for one dimension (e.g. column) detected by correct parity for all other dimensions (e.g. rows)

Double error detection: Need to add another parity bit, covering both the row and column parity bits, to distinguish one row and one column parity error from single payload error.

For an $m \times n$ array, redundancy $= R = \frac{mn}{(m-1)(n-1)}$

Minimize redundancy by:
- Choosing $m=n$.
- Increasing $m$ & $n$, but this increases chance of multiple undetectable errors.
Outline
Hamming codes: Rough† idea

If we can localise the error to one bit, then we can correct the error
1111110101

Can we use parity to provide information to guide a binary search for the error?
Yes!: One parity bit covers half the transmission (payload + parity) – if that parity bit is right, the error is in the other half.
1111110101p1p2p3p4

One parity bit covers ¼ of the transmission
Another parity bit covers 1/8th of the transmission...

† Very rough – This slide only motivates the idea, rather than proposing a scheme that would work.
Hamming codes

An $n-k$ check bits protect $2^{n-k}-(n-k)-1$ payload bits, e.g., 4 protect 11. All bits whose offset is a power of 2 (e.g., bit 1, 2, 4...) are check bits. The value of check bit $2^{b-1}$ is chosen to ensure even parity over all bits whose position has a 1 in bit $b$, e.g., bit 1 ($2^0$) covers odd bits.

If parity checks fail, and there is only one error, number (“syndrome”) formed using digits reflecting parity outcome (0=OK, 1=fail) indicates offset of error, e.g.:

Parity is (faulty bits in italics):

Faulty parity bits as ones $\Rightarrow$ error is in bit 6

Hamming codes are used in 802.16 Wireless networks.
Use of a Hamming code to correct burst errors

<table>
<thead>
<tr>
<th>Char.</th>
<th>ASCII</th>
<th>Check bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1001000</td>
<td>00110010000</td>
</tr>
<tr>
<td>a</td>
<td>1100001</td>
<td>10111001001</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>m</td>
<td>1101101</td>
<td>11101010101</td>
</tr>
<tr>
<td>i</td>
<td>1101001</td>
<td>01101011001</td>
</tr>
<tr>
<td>n</td>
<td>1101110</td>
<td>01101010110</td>
</tr>
<tr>
<td>g</td>
<td>1100111</td>
<td>01111001111</td>
</tr>
<tr>
<td>c</td>
<td>0100000</td>
<td>10011000000</td>
</tr>
<tr>
<td>o</td>
<td>1100011</td>
<td>11111000011</td>
</tr>
<tr>
<td>d</td>
<td>1100100</td>
<td>11111001100</td>
</tr>
<tr>
<td>e</td>
<td>1100101</td>
<td>00111000101</td>
</tr>
</tbody>
</table>

Order of bit transmission
Summary of checks covered so far

**Parity**: Exclusive-OR bits
- Detect odd numbers of errors

**Checksum**: Add words

**2-dimensional parity**:
- Detect odd numbers of errors
- Detect 2 errors
- Correct single error

**Hamming Codes**
- Correct single error with minimal overhead