Data Communications Fundamentals: Integrity

Cristian S. Calude Clark Thomborson

July 2010

Why check integrity?

Data can be corrupted during transmission—many factors can alter or even wipe out parts of data.

Reliable systems must have mechanisms for detecting and correcting **errors**.

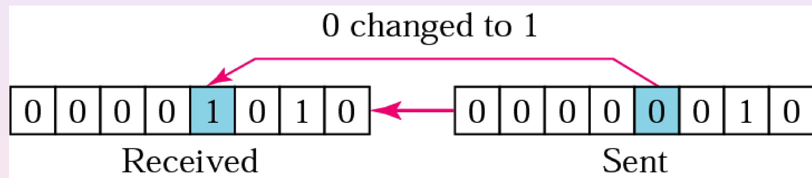
The capability to detect when a transmission has been changed is called **error detection**. In some cases a message with errors is discarded and sent again, but not always this is possible (e.g. in real-time viewing). In the later case the error has to be fixed (in real-time) and the mechanism doing this job is called **error correction**.

Types of errors

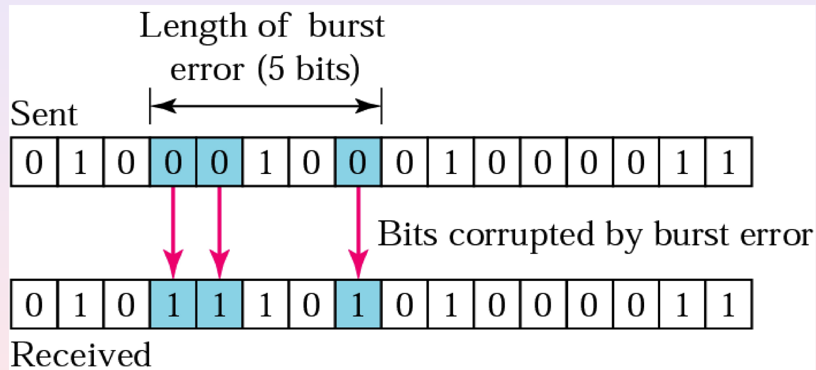
There are two types of errors:

1. single-bit error, when only one bit in the data has changed,
2. burst error, when two or more bits in the data have changed.

Single-error bit



Burst error

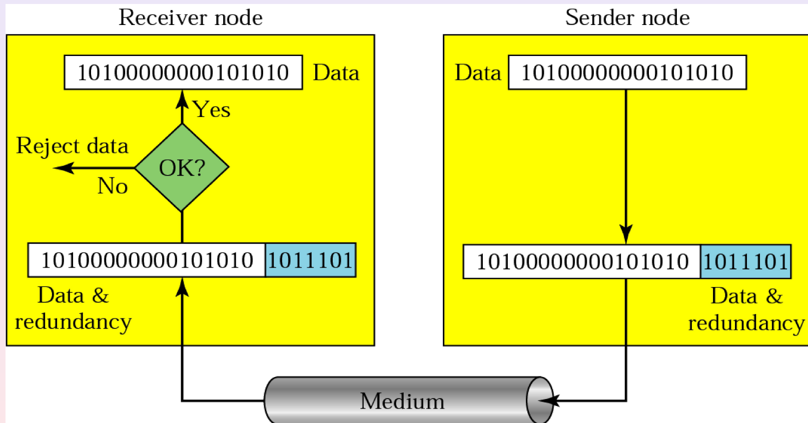


Redundancy 1

One way to check errors is by sending every data unit twice. A bit-by-bit comparison between the two versions is likely to identify all errors. The system is reasonably accurate (assuming that the errors are randomly distributed), but inefficient. Transmission time is doubled, and storage costs have increased significantly (because the transmitter and receiver must both retain a complete copy of the message).

A better approach is to add a fixed extra information to each segment (packet) of the message—this technique is called **redundancy**. There is no additional information in these error-checking bits, so they can be discarded after the check is completed. Note: natural languages are redundant, for example it is possible to understand most English sentences if vowels are omitted: FCTSSTRNGRTHNFCTN.

Redundancy 2



Redundancy 3

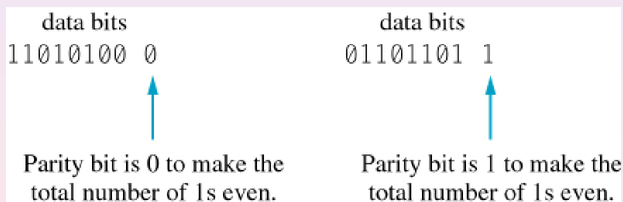
There are tree main types of redundancy checks:

1. parity check,
2. checksums,
3. cyclic redundancy check (CRC).

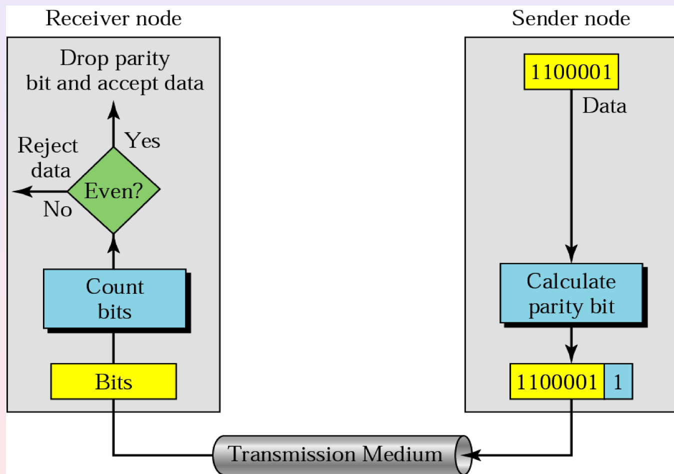
Low-density parity-check (LDPC) codes were the first to allow data transmission rates close to the theoretical maximum, the Shannon Limit.

Parity check: one-dimensional 1

A redundant bit, called **parity bit**, is added to every data unit so that the total number of 1s in the unit (including the parity bit) becomes even (or odd).



Parity check: one-dimensional 2



Parity check: one-dimensional 3

A simple parity check will detect all single-bit errors.

It can detect multiple-bit errors only if the total number of errors is odd.

Question: If the bit error rate (BER) is 0.1%, and errors are equiprobable at each bit in a 999-bit message with a single parity bit, what is the probability of an undetected error?

Parity check: one-dimensional 4

Answer: find someone who knows a little probability theory.

They should calculate the probability of no error as $(99.9\%)^{1000} = 37\%$, the probability of a single error as $1000(0.1\%)(99.9\%)^{999} = 37\%$, the probability of a double error as $\binom{1000}{2}(0.1\%)^2(99.9\%)^{998} = 18\%$, the probability of a triple error as $\binom{1000}{3}(0.1\%)^3(99.9\%)^{997} = 6\%$, and the probability of a quadruple error as 2%, and the probability of more than four errors as less than 1%.

The chance of an undetected error is thus about 20% for 999-bit messages with a single parity bit, on a channel with a BER of 0.1%.

Would you be happy with a digital communication system that introduces errors in 20% of your messages?

Parity check: two-dimensional 1

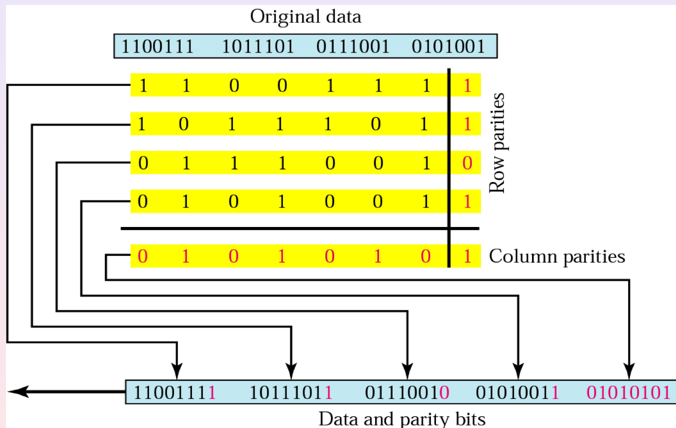
In a **two-dimensional parity check** a block of bits is organised in a table and parity is checked on both dimensions.

- First one calculates the parity bit for each data unit.
- Second one calculates the parity bit for each column and one creates a new row of 8 bits—the parity bits for the whole block.

A two-dimensional parity check significantly increases the likelihood that a burst error will be detected.

Definition. If a message $m_1 m_2 \dots m_n$ contains a single **burst error** of length B, then its bit-errors are confined to a single subsequence $m_i \dots m_{i+B-1}$ of length B.

Parity check: two-dimensional 2



Checksums 1

The method divides all data bits into 32-bit groups and treats each as an integer value. The sum of all these values gives the **checksum**. Any overflow that requires more than 32 bits is ignored.

An extra 32 bits representing in binary the checksum is then appended to the data before transmission.

The receiver divides the data bits into 32-bit groups and performs the same calculation. If the calculated checksum is different to the value received in the checksum field, then an error has occurred.

Checksums 2

A checksum is more sensitive to errors than a single-bit parity code, but it does not detect all possible errors. For example, a checksum is insensitive to any errors which *simultaneously* increases (by any value c) one of the 32-bit groups while also decreasing another 32-bit group by the same value c.

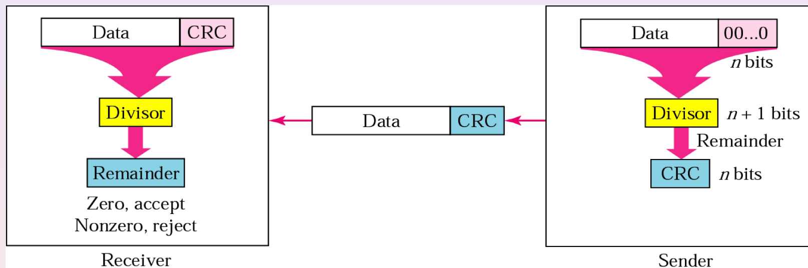
Cyclic redundancy check 1

The most powerful (and elaborate) redundancy checking technique is the **cyclic redundancy check (CRC)**.

CRC is based on binary division: a string of redundant bits—called the **CRC** or the **CRC remainder**—is appended to the end of the data unit such that the resulting data unit is exactly divisible by a second, predetermined binary number.

The receiver divides the incoming data by the same number. If the remainder is zero, the unit of data is accepted; otherwise it is rejected.

Cyclic redundancy check 2



Error correction

Error-correcting codes include so much redundant information with the unit block that the receiver is able to deduce, with high likelihood, not only how many bits are in error, but also which bits are incorrect. After these bits are inverted, all errors are corrected in the received message.

These codes are used when the error rate is high (e.g. on a WiFi channel). Detecting an error and than re-transmitting the message is very inefficient on a noisy channel, because every retransmission is likely to have errors.

Note that we cannot absolutely guarantee error correction, unless (somehow) we can place an upper bound on the total number of bit-errors in a message. In particular, if all of the error-check bits may be simultaneously in error, then any error in the data bits might not be corrected—and it may not even be detected.

Forward error correction 1

We examine the simplest case, i.e. single-bit errors. A single additional bit can detect single-bit errors. Is it enough for correction?

To correct a single-bit error in an ASCII character we must determine which of the 7 bits has changed. There are **eight** possible situations: no error, error on the first bit, error on the second bit, . . . , error on the seventh bit. Apparently we need **three** bits to code the above eight cases. But what if an error occurs in the redundancy bits themselves?

Forward error correction 2

Clearly, the number of redundancy (or, error control) bits required to correct n bits of data **cannot be constant, it depends on** n.

To calculate the number of redundancy bits r required to correct a n bits of data we note that:

- with n bits of data and r bits of redundancy we get a code of length $n + r$,
- r must be able to handle at least $n + r + 1$ different states, one for no error, $n + r$ for each possible position,
- therefore $2^r \geq n + r + 1$.

Forward error correction 3

In fact we can choose r to be the smallest integer such that $2^r \geq n + r + 1$. Here are some examples:

Number of data bits (n)	Minimum number of redundancy bits (r)	Total bits ($n + r$)
1	2	3
2	3	5
3	3	6
4	3	7
5	4	9
6	4	10
7	4	11

The Hamming code 1

The **Hamming code** is a practical solution which detects and corrects all single-bit errors in data units of any length.

A Hamming code for 7-bit ASCII code has 4 redundancy bits. These bits can be added in arbitrary positions, but it is simplest if we do this on positions 1, 2, 4, and 8. Data bits are marked with **d** and parity bits are denoted by r_8, r_4, r_2, r_1.

11	10	9	8	7	6	5	4	3	2	1
d	**d**	**d**	r_8	**d**	**d**	**d**	r_4	**d**	r_2	r_1

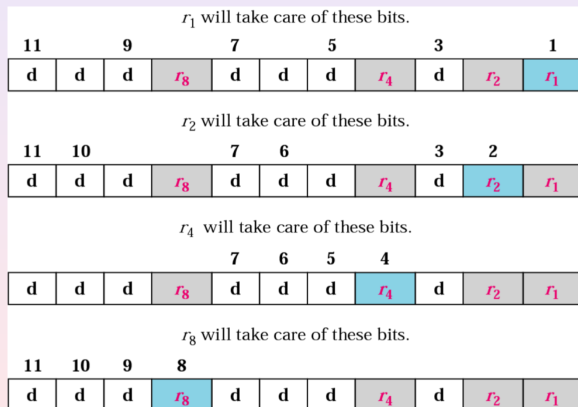
The Hamming code 2

In a Hamming code for bit-strings of length $2^n - 1$, each r bit is the parity bit for a specific combination of data bits, where the combinations follow a binary pattern shown below:

- r_1: parity on bits 1, 3, 5, 7, 9, 11, 13, 15, ..., $2^n - 1$. That is, check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc.
- r_2: parity on bits 2, 3, 6, 7, 10, 11, 14, 15, ... That is, skip 1 bit, check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc.
- r_4: parity on bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23, ... That is, skip 3 bits, check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc.
- ... $r_{2^{n-1}}$: parity on bits 2^{n-1} through $2^n - 1$. That is, skip $2^{n-1} - 1$ bits, and check 2^{n-1} bits.

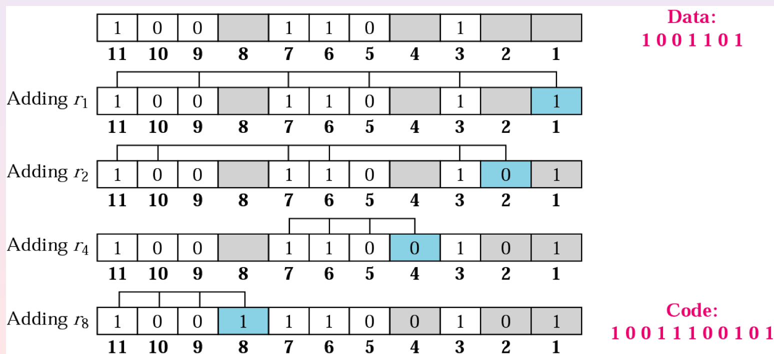
The Hamming code 3

Each data bit may be included in more than one calculation.



The Hamming code 4

This picture describes the calculation of r bits for the ASCII value of 1001101: start with data and repeatedly calculate, one by one, the parity bits r_1, r_2, r_4, r_8:

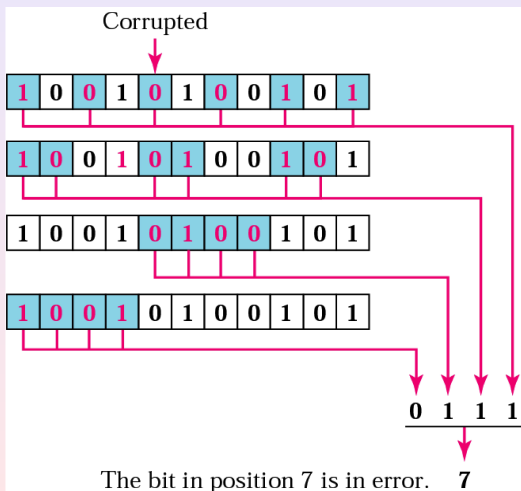


The Hamming code 5

Imagine that instead of the string 10011100101 the string 1001**0**100101 was received (the 7th bit has been changed from 1 to 0).

The receiver recalculates 4 new parity bits using the same sets of bits used by the sender plus the relevant parity r bit for each set. Then it assembles the new parity values into a binary number in the order used by the sender, r_8, r_4, r_2, r_1. This gives the location of the error bit. The sender can now reverse the value of the corrupted bit!

The Hamming code 6



Multiple-bit error correction 1

The basic Hamming code cannot correct multiple-bit errors, but can easily be adapted to cover the case where bit-errors occur in bursts.

Burst errors are common in satellite transmissions, due to sunspots and other transient phenomena which, occasionally, greatly increase the error rate for a brief period of time.

Scratches on CDs and DVDs introduce burst errors into the data read from these disks.

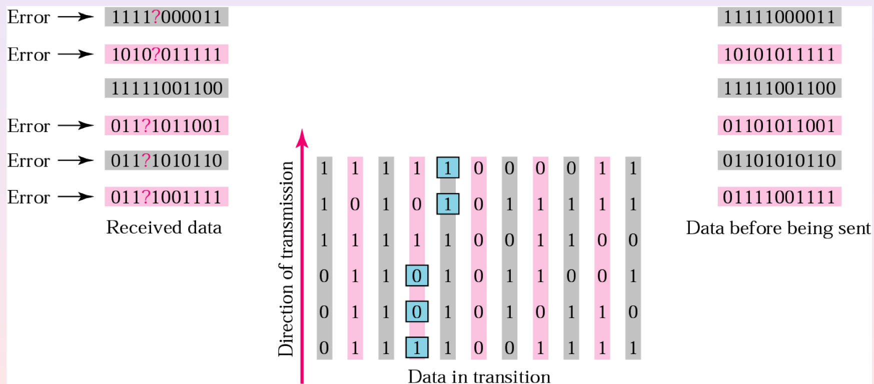
Multiple-bit error correction 2

To protect a message of length N against a single burst error of length $B \leq C$, we can break it up into groups (columns) of length C, then transmit it in transposed order with a Hamming code for each row.

No burst error of length $B < C$ can introduce more than one bit error in a row, so the row-wise Hamming codes are sufficient to correct a single burst error.

On the next slide, we show an example of the technique with $N = 36$ data bits. These bits are organised into columns of length $C = 6$, where the 6 data bits in each row are protected by a 5-bit Hamming code. A burst error of length 5 has affected two columns in our example, but has introduced at most one error in each row – so this error can be corrected by the Hamming code for the affected rows.

Multiple-bit error correction 3



Multiple-bit error correction 4

Hamming codes are pretty easy to understand, and they are pretty easy to implement – but they are not an effective way to protect very long messages against errors in arbitrary positions (i.e. multiple bit errors that aren't in bursts).

Much more powerful error-correcting codes were developed by Bose, Chaudhari, and Hocquenghem in 1959-60. These are the BCH codes. The Reed-Solomon codes (also developed in 1960) are a very important sub-class of the BCH codes. Efficient decoding algorithms, suitable for special-purpose hardware implementation, were developed in 1969; these were used in satellite communications.

Nowadays, Reed-Solomon error-correction is used routinely in compact disks and DVDs.

Multiple-bit error correction 5

A Hamming code uses $r = \log n$ bits to guard n data bits, so we can send only 2^n different messages in a Hamming-protected message of length $n + r$.

This might, or might not, be the appropriate ratio of error-checking (redundant) symbols to information-carrying symbols for this channel. Let's develop a little more theory...

Multiple-bit error correction 6

The **Hamming distance** between two m-bit strings is the number of bits on which the two strings differ.

Given a finite set of codewords one can compute its *minimum distance*, the smallest value of the distance between two codewords in the set.

If d is the minimum distance of a finite set of codewords, then the method can detect any error affecting fewer than d bits (such a change would create an invalid codeword) and correct any error affecting fewer than $d/2$ bits.

Multiple-bit error correction 7

For example, if $d = 10$, and 4 bits of a codeword were damaged, then the string cannot possibly be a valid codeword: at least 10 bits must be changed to create another valid codeword.

If the receiver assumes that any error will affect fewer than 5 bits, then she needs only to find the closest valid codeword to the received damaged one to conclude that it is the correct codeword. Indeed, any other codeword would have had at least 6 bits damaged to resemble the received string.

When designing a code that can correct $d/2$ or fewer errors in a message, we must select codewords which are at least Hamming distance d from each other. It's an interesting combinatorial puzzle... randomly-chosen codebooks do pretty well (but would require decoders to use large – and therefore slow and/or expensive – lookup tables)...

Error correction on the Internet is performed at multiple levels

- Each Ethernet frame carries a CRC-32 checksum. The receiver discards frames if their checksums do not match.
- The IPv4 header contains a header checksum of the contents of the header (excluding the checksum field). Packets with checksums that don't match are discarded.
- The checksum was omitted from the IPv6 header, because most current link layer protocols have error detection.
- UDP has an optional checksum. Packets with wrong checksums are discarded.
- TCP has a checksum of the payload, TCP header (excluding the checksum field) and source- and destination addresses of the IP header. Packets found to have incorrect checksums are discarded and will eventually be retransmitted (when the sender receives three identical ACKs, or times out).