MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2010 (Last update: October 7, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

LECTURE 7
Detecting Bit Errors

These lecture notes discuss some techniques for **error detection**. The reason why error detection is important is that no practical error correction schemes can perfectly correct all errors in a message. For example, any reasonable error correction scheme that can correct all patterns of t or fewer errors will have some error pattern of t or more errors that cannot be corrected. Our goal is not to eliminate all errors, but to reduce the bit error rate to a low enough value that the occasional corrupted coded message is not a problem: the receiver can just discard such messages and perhaps request a retransmission from the sender (we will study such retransmission protocols later in the term). To decide whether to keep or discard a message, the receiver needs a way to detect any errors that might remain after the error correction and decoding schemes have done their job: this task is done by an error detection scheme.

An error detection scheme works as follows. The sender takes the message and produces a compact *hash* or *digest* of the message, with the idea that commonly occurring corruptions of the message will cause the hash to be different from the correct value. The sender includes the hash with the message, and then passes that over to the error correcting mechanisms, which code the message. The receiver gets the coded bits, runs the error correction decoding steps, and then obtains the presumptive set of original message bits and the hash. The receiver computes the same hash over the presumptive message bits and compares the result with the presumptive hash it has decoded. If the results disagree, then clearly there has been some unrecoverable error, and the message is discarded. If the results agree, then the receiver believes the message to be correct. Note that if the results agree, the receiver can only *believe* the message to be correct; it is certainly possible (though, for good detection schemes, unlikely) for two different message bit sequences to have the same hash.

The topic of this lecture is the design of appropriate error detection hash functions. The design depends on the errors we anticipate. If the errors are adversarial in nature, e.g., from a malicious party who can change the bits as they are sent over the channel, then the hash function must guard against as many of the enormous number of different error patterns that might occur. This task requires cryptographic protection, and is done in practice using schemes like SHA-1, the secure hash algorithm. We won't study these in 6.02, focusing instead on non-malicious, random errors introduced when bits are sent over communication channels. The error detection hash functions in this case are typically called *checksums*: they protect against certain random forms of bit errors, but are by no means the method to use when communicating over an insecure channel. We will study two simple checksum algorithms: the Adler-32 checksum and the Cyclic Redundancy Check (CRC).[1]

■ 7.1 Adler-32 Checksum

Many checksum algorithms compute the hash of a message by adding together bits of the message. Usually the bits are collected into 8-bit bytes or 32-bit words and the addition is performed 8 or 32 bits at a time. A simple checksum isn't very robust when there is more than one error: it's all too easy for the second error to *mask* the first error, e.g., an earlier 1 that was corrupted to 0 can be offset in the sum by a later 0 corrupted to a 1. So most checksums use a formula where a bit's effect on the sum is dependent on its position in the message as well as its value.

The Adler-32 checksum is an elegant and reasonably effective check-bit computation that's particularly easy to implement in software. It works on 8-bit bytes of the message, assembled from eight consecutive message bits. Processing each byte $(D_1, D_2, \ldots, D_n)$, compute two sums A and B:

$$\begin{aligned} A &= (1 + \text{sum of the bytes}) \mod 65521 \\ &= (1 + D_1 + D_2 + \ldots + D_n) \mod 65521 \end{aligned}$$

$$\begin{aligned} B &= (\text{sum of the A values after adding each byte}) \mod 65521 \\ &= ((1 + D_1) + (1 + D_1 + D_2) + \ldots + (1 + D_1 + D_2 + \ldots + D_n)) \mod 65521 \end{aligned}$$

After the modulo operation the A and B values can be represented as 16-bit quantities. The Adler-32 checksum is the 32-bit quantity $(B \ll 16) + A$.

The Adler-32 checksum requires messages that are several hundred bytes long before it reaches its full effectiveness, i.e., enough bytes so that A exceeds 65521.[2] Methods like the Adler-32 checksum are used to check whether large files being transferred have errors; Adler-32 itself is used in the popular zlib compression utility and (in rolling window form) in the rsync file synchronization program.

For network packet transmissions, typical sizes range between 40 bytes and perhaps 10000 bytes, and often packets are on the order of 1000 bytes. For such sizes, a more effective error detection method is the **cyclic redundancy check (CRC)**. CRCs work well over shorter messages and are easy to implement in hardware using shift registers. For these reasons, they are extremely popular.

[1]Sometimes, the literature uses "checksums" to mean something different from a "CRC", using checksums for methods that involve the addition of groups of bits to produce the result, and CRCs for methods that involve polynomial division. We use the term "checksum" to include both kinds of functions, which are both applicable to random errors and not to insecure channels (unlike secure hash functions.

[2]65521 is the largest prime smaller than 2^{16}. It is not clear to what extent the primality of the modulus matters, and some studies have shown that it doesn't seem to matter much.

■ 7.2 Cyclic Redundancy Check

A CRC is an example of a block code, but it can operate on blocks of any size. Given a message block of size k bits, it produces a compact digest of size r bits, where r is a constant (typically between 8 and 32 bits in real implementations). Together, the $k + r = n$ bits constitute a **code word**. Every valid code word has a certain minimum Hamming distance from every other valid code word to aid in error detection.

A CRC is an example of a *polynomial code* as well as an example of a *cyclic code*. The idea in a polynomial code is to represent every code word $w = w_{n-1}w_{n-2}w_{n-2}\dots w_0$ as a polynomial of degree $n-1$. That is, we write

$$w(x) = \sum_{i=0}^{n-1} w_i x^i. \tag{7.1}$$

For example, the code word 11000101 may be represented as the polynomial $x^7 + x^6 + x^2 + 1$, plugging the bits into Eq.(7.1).

We use the term *code polynomial* to refer to the polynomial corresponding to a code word.

The key idea in a CRC (and, indeed, in any cyclic code) is to ensure that *every valid code polynomial is a multiple of a generator polynomial*, $g(x)$. We will look at the properties of good generator polynomials in a bit, but for now let's look at some properties of codes built with this property.

All arithmetic in our CRC will be done in $\mathbb{F}_2$. The normal rules of polynomial addition, division, multiplication, and division apply, except that all coefficients are either 0 or 1 and the coefficients add and multiply using the $\mathbb{F}_2$ rules. In particular, note that all minus signs can be replaced with + signs, making life quite convenient.

■ 7.2.1 Encoding Step

The CRC encoding step of producing the digest is simple. Given a message, construct the message polynomial $m(x)$ using the same method as Eq.(7.1). Then, our goal is to construct the code polynomial, $w(x)$ from $m(x)$ and $g(x)$ so that $g(x)$ divides $w(x)$ (i.e., $w(x)$ is a multiple of $g(x)$).

First, let us multiply $m(x)$ by x^{n-k}. The reason we do this multiplication is to shift the message left by $n-k$ bits, so we can add the redundant check bits ($n-k$ of them) so that the code word is in systematic form. It should be easy to verify that this multiplication produces a polynomial whose coefficients correspond to original message bits followed by all zeroes (for the check bits we're going to add in below).

Then, let's divide $x^{n-k}m(x)$ by $g(x)$. If the remainder from the polynomial division is 0, then we have a valid codeword. Otherwise, we have a remainder, $\mathcal{R}$. We know that if we subtract this remainder from the polynomial $x^{n-k}m(x)$, we will obtain a new polynomial that will be a multiple of $g(x)$. Remembering that we are in $\mathbb{F}_2$, we can replace the subtraction with an addition, getting:

$$w(x) = x^{n-k}m(x) + R\{x^{n-k}m(x)/g(x)\}, \tag{7.2}$$

where the notation $R\{a(x)/b(x)\}$ stands for the remainder when $a(x)$ is divided by $b(x)$.

The encoder is now straightforward to define. Take the message, construct the message

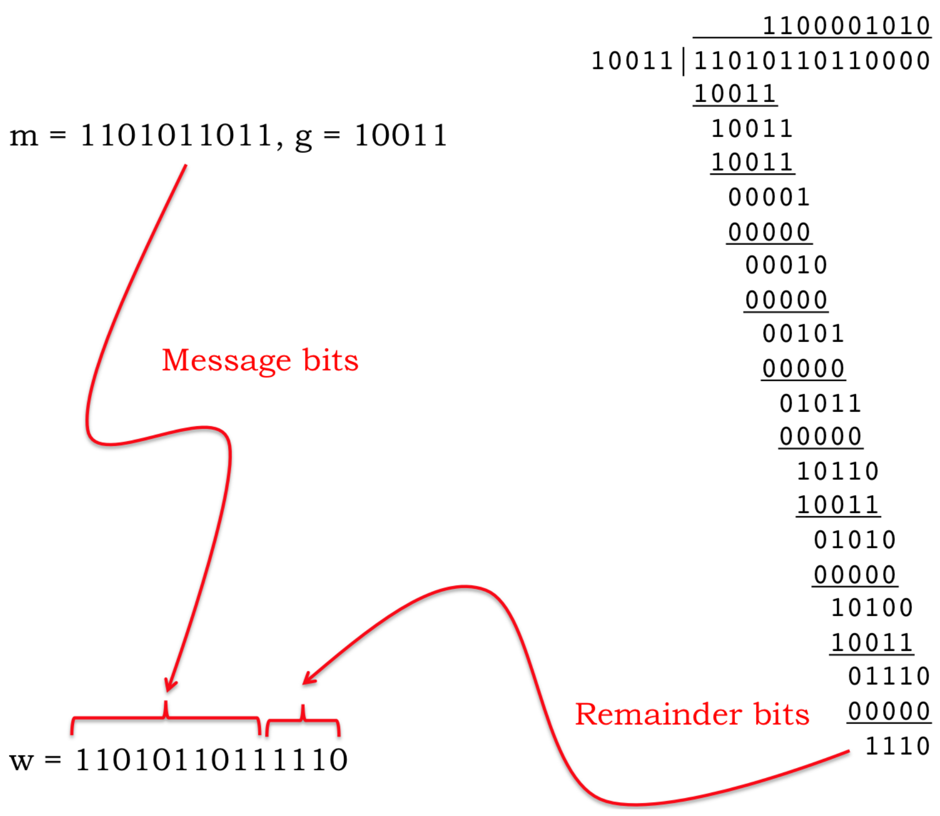

Figure 7-1: CRC computations using "long division".

polynomial, multiply by x^{n-k}, and then divide that by $g(x)$. The remainder forms the check bits, acting as the digest for the entire message. Send these bits appended to the message.

7.2.2 Decoding Step

The decoding step is essentially identical to the encoding step, one of the advantages of using a CRC. Separate each code word received into the message and remainder portions, and verify whether the remainder calculated from the message matches the bits sent together with the message. A mismatch guarantees that an error has occurred; a match suggests a reasonable likelihood of the message being correct, *as long as a suitable generator polynomial is used*.

7.2.3 Mechanics of division

There are several efficient ways to implement the division and remaindering operations needed in a CRC computation. The schemes used in practice essentially mimic the "long division" strategies one learns in elementary school. Figure 7-1 shows an example to refresh your memory!

■ 7.2.4 Good Generator Polynomials

So how should one pick good generator polynomials? There is no magic prescription here, but what commonly occuring error patterns do to the received code words, we can form some guidelines. To develop suitable properties for $g(x)$, first observe that if the receiver gets a bit sequence, we can think of it as the code word sent added to a sequence of zero or more errors. That is, take the bits obtained by the receiver and construct a polynomial, $r(x)$ from it. We can think of $r(x)$ as being the sum of $w(x)$, which is what the sender sent (the receiver doesn't know what the real w was) and an *error polynomial*, $e(x)$.

Here's the key point: If $r(x) = w(x) + e(x)$ is *not* a multiple of $g(x)$, then the receiver is *guaranteed* to detect the error. Because $w(x)$ is constructed as a multiple of $g(x)$, this statement is the same as saying that if $e(x)$ is not a multiple of $g(x)$, the receiver is guaranteed to detect the error. On the other hand, if $r(x)$, and therefore $e(x)$, *is* a multiple of $g(x)$, then we either have no errors, or we have an error that we cannot detect (i.e., an erroneous reception that we falsely identify as correct). Our goal is to ensure that this situation does not happen for commonly occurring error patterns.

1. First, note that for single error patterns, $e(x) = x^i$ for some i. That means we must ensure that $g(x)$ has at least two terms.

2. Suppose we want to be able to detect all error patterns with two errors. That error pattern may be written as $x^i + x^j = x^i(1 + x^{j-i})$, for some i and $j > i$. If $g(x)$ does not divide this term, then the resulting CRC can detect all double errors.

3. Now suppose we want to detect all odd numbers of errors. If $(1 + x)$ is a factor of $g(x)$, then $g(x)$ must have an *even number of terms*. The reason is that any polynomial with coefficients in $\mathbb{F}_2$ of the form $(1 + x)h(x)$ must evaluate to 0 when we set x to 1. If we expand $(1 + x)h(x)$, if the answer must be 0 when $x = 1$, the expansion must have an even number of terms. Therefore, if we make $1 + x$ a factor of $g(x)$, the resulting CRC will be *able to detect all error patterns with an odd number of errors*. Note, however, that the converse statement is not true: a CRC may be able to detect an odd number of errors even when its $g(x)$ is not a multiple of $(1 + x)$. But all CRCs used in practice do have $(1 + x)$ as a factor because its the simplest way to achieve this goal.

4. Another guideline used by some CRC schemes in practice is the ability to detect *burst errors*. Let us define a burst error pattern of length b as a sequence of bits $1\varepsilon_{b-2}\varepsilon_{b-3}\ldots\varepsilon_1 1$: that is, the number of bits is b, the first and last bits are both 1, and the bits ε_i in the middle could be either 0 or 1. The minimum burst length is 2, corresponding to the pattern "11".

   Suppose we would like our CRC to detect all such error patterns, where $e(x) = x^s(1 \cdot x^{b-1} + \sum_{i=1}^{b-2} \varepsilon_i x^i + 1)$. This polynomial represents a burst error pattern of size b starting s bits to the left from the end of the packet. If we pick $g(x)$ to be a polynomial of degree b, and if $g(x)$ does not have x as a factor, then any error pattern of length $\leq b$ is guaranteed to be detected, because $g(x)$ will not divide a polynomial of degree smaller than its own. Moreover, there is exactly one error pattern of length $b + 1$—corresponding to the case when the burst error pattern matches the coefficients of $g(x)$ itself—that will not be detected. All other error patterns of length $b + 1$ will be detected by this CRC.

CRC-1: $x + 1$ (parity bit)
CRC-5-EPC: $x^5 + x^3 + 1$ (Gen 2 RFID)
CRC-8-WCDMA: $x^8 + x^7 + x^4 + x^3 + x + 1$
CRC-15-CAN: $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$ (Controller Area Network in vehicles)
CRC-16-ANSI: $x^{16} + x^{15} + x^2 + 1$ (USB, etc.)
CRC-16-CCITT: $x^{16} + x^{12} + x^5 + 1$ (Bluetooth, etc.)
CRC-16-DECT: $x^{16} + x^{10} + x^8 + x^7 + x^3 + 1$ (cordless phones)
CRC-32-IEEE: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Ethernet, WiFi, POSIX cksum, etc.)

Figure 7-2: Commonly used CRC generator polynomials, $g(x)$. From Wikipedia.

If fact, such a CRC is quite good at detecting longer burst errors as well, though it cannot detect all of them.

CRCs are examples of *cyclic* codes, which have the property that if c is a code word, then any cyclic shift (rotation) of c is another valid code word. Hence, referring to Eq.(7.1), we find that one can represent the polynomial corresponding to one cyclic left shift of w as

$$\begin{aligned} w^{(1)}(x) &= w_{n-1} + w_0 x + w_1 x^2 + \ldots w_{n-2}x^{n-1} && (7.3) \\ &= xw(x) + (1 + x^n)w_{n-1} && (7.4) \end{aligned}$$

Now, because $w^{(1)}(x)$ must also be a valid code word, it must be a multiple of $g(x)$, which means that $g(x)$ must divide $1 + x^n$. Note that $1 + x^n$ corresponds to a double error pattern; what this observation implies is that the CRC scheme using cyclic code polynomials can detect the errors we want to detect (such as all double bit errors) as long as $g(x)$ is picked so that the smallest n for which $1 + x^n$ is a multiple of $g(x)$ is quite large. For example, in practice, a common 16-bit CRC has a $g(x)$ for which the smallest such value of n is $2^{15} - 1 = 32767$, which means that it's quite effective for all messages of length smaller than that.

■ 7.2.5 CRCs in practice

CRCs are used in essentially all communication systems. The table in Figure 7-2, culled from Wikipedia, has a list of common CRCs and practical systems in which they are used. You can see that they all have an even number of terms, and verify (if you wish) that $1 + x$ divides each of them.