***2.4.2. Internet Checksum Algorithm***

A second approach to error detection is exemplified by the Internet checksum. Although it is not used at the link level, it nevertheless provides the same sort of functionality as CRCs and parity, so we discuss it here. We will see examples of its use in [Section 3.2](https://learning.oreilly.com/library/view/computer-networks-5th/9780123850591/B978012385059100003X-08.xhtml#s0070), [Section 5.1](https://learning.oreilly.com/library/view/computer-networks-5th/9780123850591/B9780123850591000053.xhtml#s0010) and [Section 5.2](https://learning.oreilly.com/library/view/computer-networks-5th/9780123850591/B9780123850591000053.xhtml#s0015).

The idea behind the Internet checksum is very simple—you add up all the words that are transmitted and then transmit the result of that sum. The result is the checksum. The receiver performs the same calculation on the received data and compares the result with the received checksum. If any transmitted data, including the checksum itself, is corrupted, then the results will not match, so the receiver knows that an error occurred.

You can imagine many different variations on the basic idea of a checksum. The exact scheme used by the Internet protocols works as follows. Consider the data being checksummed as a sequence of 16-bit integers. Add them together using 16-bit ones complement arithmetic (explained below) and then take the ones complement of the result. That 16-bit number is the checksum.

In ones complement arithmetic, a negative integer (− *x*) is represented as the complement of *x*; that is, each bit of *x* is inverted. When adding numbers in ones complement arithmetic, a carryout from the most significant bit needs to be added to the result. Consider, for example, the addition of −5 and −3 in ones complement arithmetic on 4-bit integers: +5 is 0101, so −5 is 1010; +3 is 0011, so −3 is 1100. If we add 1010 and 1100, ignoring the carry, we get 0110. In ones complement arithmetic, the fact that this operation caused a carry from the most significant bit causes us to increment the result, giving 0111, which is the ones complement representation of −8(obtained by inverting the bits in 1000), as we would expect.

The following routine gives a straightforward implementation of the Internet's checksum algorithm. The **count** argument gives the length of **buf** measured in 16-bit units. The routine assumes that **buf** has already been padded with 0s to a 16-bit boundary.

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);

}

This code ensures that the calculation uses ones complement arithmetic rather than the twos complement that is used in most machines. Note the **if** statement inside the **while** loop. If there is a carry into the top 16 bits of **sum**, then we increment **sum** just as in the previous example.

Compared to our repetition code, this algorithm scores well for using a small number of redundant bits—only 16 for a message of any length—but it does not score extremely well for strength of error detection. For example, a pair of single-bit errors, one of which increments a word and one of which decrements another word by the same amount, will go undetected. The reason for using an algorithm like this in spite of its relatively weak protection against errors (compared to a CRC, for example) is simple: This algorithm is much easier to implement in software. Experience in the ARPANET suggested that a checksum of this form was adequate. One reason it is adequate is that this checksum is the last line of defense in an end-to-end protocol; the majority of errors are picked up by stronger error detection algorithms, such as CRCs, at the link level.