Network Working Group
Request for Comments: 1071

R. Braden
ISI
D. Borman
Cray Research
C. Partridge
BBN Laboratories
September 1988

Computing the Internet Checksum

Status of This Memo

This memo summarizes techniques and algorithms for efficiently computing the Internet checksum. It is not a standard, but a set of useful implementation techniques. Distribution of this memo is unlimited.

1. Introduction

This memo discusses methods for efficiently computing the Internet checksum that is used by the standard Internet protocols IP, UDP, and TCP.

An efficient checksum implementation is critical to good performance. As advances in implementation techniques streamline the rest of the protocol processing, the checksum computation becomes one of the limiting factors on TCP performance, for example. It is usually appropriate to carefully hand-craft the checksum routine, exploiting every machine-dependent trick possible; a fraction of a microsecond per TCP data byte can add up to a significant CPU time savings overall.

In outline, the Internet checksum algorithm is very simple:

(1) Adjacent octets to be checksummed are paired to form 16-bit integers, and the 1's complement sum of these 16-bit integers is formed.

(2) To generate a checksum, the checksum field itself is cleared, the 16-bit 1's complement sum is computed over the octets concerned, and the 1's complement of this sum is placed in the checksum field.

(3) To check a checksum, the 1's complement sum is computed over the same set of octets, including the checksum field. If the result is all 1 bits (-0 in 1's complement arithmetic), the check succeeds.

Suppose a checksum is to be computed over the sequence of octets

A, B, C, D, ... , Y, Z. Using the notation [a,b] for the 16-bit integer a*256+b, where a and b are bytes, then the 16-bit 1's complement sum of these bytes is given by one of the following:

[A,B] +' [C,D] +' ... +' [Y,Z] [1]

[A,B] +' [C,D] +' ... +' [Z,0] [2]

where +' indicates 1's complement addition. These cases correspond to an even or odd count of bytes, respectively.

On a 2's complement machine, the 1's complement sum must be computed by means of an "end around carry", i.e., any overflows from the most significant bits are added into the least significant bits. See the examples below.

Section 2 explores the properties of this checksum that may be exploited to speed its calculation. Section 3 contains some numerical examples of the most important implementation techniques. Finally, Section 4 includes examples of specific algorithms for a variety of common CPU types. We are grateful to Van Jacobson and Charley Kline for their contribution of algorithms to this section.

The properties of the Internet checksum were originally discussed by Bill Plummer in IEN-45, entitled "Checksum Function Design". Since IEN-45 has not been widely available, we include it as an extended appendix to this RFC.

2. Calculating the Checksum

This simple checksum has a number of wonderful mathematical properties that may be exploited to speed its calculation, as we will now discuss.

(A) Commutative and Associative

As long as the even/odd assignment of bytes is respected, the sum can be done in any order, and it can be arbitrarily split into groups.

For example, the sum [1] could be split into:

([A,B] +' [C,D] +' ... +' [J,0])

+' ([0,K] +' ... +' [Y,Z]) [3]

(B) Byte Order Independence

The sum of 16-bit integers can be computed in either byte order. Thus, if we calculate the swapped sum:

[B,A] +' [D,C] +' ... +' [Z,Y] [4]

the result is the same as [1], except the bytes are swapped in the sum! To see why this is so, observe that in both orders the carries are the same: from bit 15 to bit 0 and from bit 7 to bit 8. In other words, consistently swapping bytes simply rotates the bits within the sum, but does not affect their internal ordering.

Therefore, the sum may be calculated in exactly the same way regardless of the byte order ("big-endian" or "little-endian") of the underlaying hardware. For example, assume a "little-endian" machine summing data that is stored in memory in network ("big-endian") order. Fetching each 16-bit word will swap bytes, resulting in the sum [4]; however, storing the result back into memory will swap the sum back into network byte order.

Byte swapping may also be used explicitly to handle boundary alignment problems. For example, the second group in [3] can be calculated without concern to its odd/even origin, as:

[K,L] +' ... +' [Z,0]

if this sum is byte-swapped before it is added to the first group. See the example below.

(C) Parallel Summation

On machines that have word-sizes that are multiples of 16 bits, it is possible to develop even more efficient implementations. Because addition is associative, we do not have to sum the integers in the order they appear in the message. Instead we can add them in "parallel" by exploiting the larger word size.

To compute the checksum in parallel, simply do a 1's complement addition of the message using the native word size of the machine. For example, on a 32-bit machine we can add 4 bytes at a time: [A,B,C,D]+'... When the sum has been computed, we "fold" the long sum into 16 bits by adding the 16-bit segments. Each 16-bit addition may produce new end-around carries that must be added.

Furthermore, again the byte order does not matter; we could instead sum 32-bit words: [D,C,B,A]+'... or [B,A,D,C]+'... and then swap the bytes of the final 16-bit sum as necessary. See the examples below. Any permutation is allowed that collects

all the even-numbered data bytes into one sum byte and the odd-numbered data bytes into the other sum byte.

There are further coding techniques that can be exploited to speed up the checksum calculation.

(1) Deferred Carries

Depending upon the machine, it may be more efficient to defer adding end-around carries until the main summation loop is finished.

One approach is to sum 16-bit words in a 32-bit accumulator, so the overflows build up in the high-order 16 bits. This approach typically avoids a carry-sensing instruction but requires twice as many additions as would adding 32-bit segments; which is faster depends upon the detailed hardware architecture.

(2) Unwinding Loops

To reduce the loop overhead, it is often useful to "unwind" the inner sum loop, replicating a series of addition commands within one loop traversal. This technique often provides significant savings, although it may complicate the logic of the program considerably.

(3) Combine with Data Copying

Like checksumming, copying data from one memory location to another involves per-byte overhead. In both cases, the bottleneck is essentially the memory bus, i.e., how fast the data can be fetched. On some machines (especially relatively slow and simple micro-computers), overhead can be significantly reduced by combining memory-to-memory copy and the checksumming, fetching the data only once for both.

(4) Incremental Update

Finally, one can sometimes avoid recomputing the entire checksum when one header field is updated. The best-known example is a gateway changing the TTL field in the IP header, but there are other examples (for example, when updating a source route). In these cases it is possible to update the checksum without scanning the message or datagram.

To update the checksum, simply add the differences of the sixteen bit integers that have been changed. To see why this works, observe that every 16-bit integer has an additive inverse and that addition is associative. From this it follows that given the original value m, the new value m', and the old

checksum C, the new checksum C′ is:

$$C' = C + (-m) + m' = C + (m' - m)$$

3. Numerical Examples

We now present explicit examples of calculating a simple 1's complement sum on a 2's complement machine. The examples show the same sum calculated byte by bye, by 16-bits words in normal and swapped order, and 32 bits at a time in 3 different orders. All numbers are in hex.

	Byte-by-byte		"Normal" Order	Swapped Order
Byte 0/1:	00	01	0001	0100
Byte 2/3:	f2	03	f203	03f2
Byte 4/5:	f4	f5	f4f5	f5f4
Byte 6/7:	f6	f7	f6f7	f7f6
Sum1:	2dc	1f0	2ddf0	1f2dc
	dc	f0	ddf0	f2dc
Carrys:	1	2	2	1
Sum2:	dd	f2	ddf2	f2dd
Final Swap:	dd	f2	ddf2	ddf2

Byte 0/1/2/3:	0001f203	010003f2	03f20100
Byte 4/5/6/7:	f4f5f6f7	f5f4f7f6	f7f6f5f4
Sum1:	0f4f7e8fa	0f6f4fbe8	0fbe8f6f4
Carries:	0	0	0
Top half:	f4f7	f6f4	fbe8
Bottom half:	e8fa	fbe8	f6f4
Sum2:	1ddf1	1f2dc	1f2dc
	ddf1	f2dc	f2dc
Carrys:	1	1	1
Sum3:	ddf2	f2dd	f2dd
Final Swap:	ddf2	ddf2	ddf2

Finally, here an example of breaking the sum into two groups, with the second group starting on a odd boundary:

```
                Byte-by-byte    Normal
                                 Order

     Byte 0/1:    00   01        0001
     Byte 2/ :    f2  (00)       f200
                 ---  ---       -----
     Sum1:        f2   01        f201

     Byte 4/5:    03   f4        03f4
     Byte 6/7:    f5   f6        f5f6
     Byte 8/:     f7  (00)       f700
                 ---  ---       -----
     Sum2:                      1f0ea

     Sum2:                       f0ea
     Carry:                         1
                               -----
     Sum3:                       f0eb

     Sum1:                       f201
     Sum3 byte swapped:          ebf0
                               -----
     Sum4:                      1ddf1

     Sum4:                       ddf1
     Carry:                         1
                               -----
     Sum5:                       ddf2
```

4. Implementation Examples

In this section we show examples of Internet checksum implementation algorithms that have been found to be efficient on a variety of CPU's. In each case, we show the core of the algorithm, without including environmental code (e.g., subroutine linkages) or special-case code.

4.1 "C"

The following "C" code algorithm computes the checksum with an inner loop that sums 16-bits at a time in a 32-bit accumulator.

```
in 6
    {
        /* Compute Internet Checksum for "count" bytes
         *         beginning at location "addr".
         */
    register long sum = 0;

     while( count > 1 )  {
        /*  This is the inner loop */
            sum += * (unsigned short) addr++;
            count -= 2;
    }

        /*  Add left-over byte, if any */
    if( count > 0 )
            sum += * (unsigned char *) addr;

        /*  Fold 32-bit sum to 16 bits */
    while (sum>>16)
        sum = (sum & 0xffff) + (sum >> 16);

    checksum = ~sum;
}
```

4.2 Motorola 68020

The following algorithm is given in assembler language for a Motorola 68020 chip. This algorithm performs the sum 32 bits at a time, and unrolls the loop with 16 replications. For clarity, we have omitted the logic to add the last fullword when the length is not a multiple of 4. The result is left in register d0.

With a 20MHz clock, this routine was measured at 134 usec/kB summing random data. This algorithm was developed by Van Jacobson.

```
    movl    d1,d2
    lsrl    #6,d1       | count/64 = # loop traversals
    andl    #0x3c,d2    | Then find fractions of a chunk
    negl    d2
    andb    #0xf,cc     | Clear X (extended carry flag)

    jmp     pc@(2$-.-2:b,d2)  | Jump into loop

1$:     | Begin inner loop...

    movl    a0@+,d2     |  Fetch 32-bit word
    addxl   d2,d0       |    Add word + previous carry
    movl    a0@+,d2     |  Fetch 32-bit word
    addxl   d2,d0       |    Add word + previous carry

        | ... 14 more replications
2$:
    dbra    d1,1$   | (NB- dbra doesn't affect X)

    movl    d0,d1   | Fold 32 bit sum to 16 bits
    swap    d1      | (NB- swap doesn't affect X)
    addxw   d1,d0
    jcc     3$
    addw    #1,d0
3$:
    andl    #0xffff,d0
```

4.3 Cray

The following example, in assembler language for a Cray CPU, was contributed by Charley Kline. It implements the checksum calculation as a vector operation, summing up to 512 bytes at a time with a basic summation unit of 32 bits. This example omits many details having to do with short blocks, for clarity.

Register A1 holds the address of a 512-byte block of memory to checksum. First two copies of the data are loaded into two vector registers. One is vector-shifted right 32 bits, while the other is vector-ANDed with a 32 bit mask. Then the two vectors are added together. Since all these operations chain, it produces one result per clock cycle. Then it collapses the result vector in a loop that adds each element to a scalar register. Finally, the end-around carry is performed and the result is folded to 16-bits.

```
      EBM
      A0      A1
      VL      64            use full vectors
      S1      <32           form 32-bit mask from the right.
      A2      32
      V1      ,A0,1            load packet into V1
      V2      S1&V1            Form right-hand 32-bits in V2.
      V3      V1>A2            Form left-hand 32-bits in V3.
      V1      V2+V3            Add the two together.
      A2      63            Prepare to collapse into a scalar.
      S1      0
      S4      <16           Form 16-bit mask from the right.
      A4      16
CK$LOOP S2    V1,A2
      A2      A2-1
      A0      A2
      S1      S1+S2
      JAN     CK$LOOP
      S2      S1&S4           Form right-hand 16-bits in S2
      S1      S1>A4           Form left-hand 16-bits in S1
      S1      S1+S2
      S2      S1&S4           Form right-hand 16-bits in S2
      S1      S1>A4           Form left-hand 16-bits in S1
      S1      S1+S2
      S1      #S1            Take one's complement
      CMR            At this point, S1 contains the checksum.
```

4.4 IBM 370

The following example, in assembler language for an IBM 370 CPU, sums the data 4 bytes at a time. For clarity, we have omitted the logic to add the last fullword when the length is not a multiple of 4, and to reverse the bytes when necessary. The result is left in register RCARRY.

This code has been timed on an IBM 3090 CPU at 27 usec/KB when summing all one bits. This time is reduced to 24.3 usec/KB if the trouble is taken to word-align the addends (requiring special cases at both the beginning and the end, and byte-swapping when necessary to compensate for starting on an odd byte).

```
*       Registers RADDR and RCOUNT contain the address and length of
*                the block to be checksummed.
*
*       (RCARRY, RSUM) must be an even/odd register pair.
*       (RCOUNT, RMOD) must be an even/odd register pair.
*
CHECKSUM  SR    RSUM,RSUM       Clear working registers.
          SR    RCARRY,RCARRY
          LA    RONE,1          Set up constant 1.
*
          SRDA  RCOUNT,6        Count/64 to RCOUNT.
          AR    RCOUNT,RONE       +1 = # times in loop.
          SRL   RMOD,26         Size of partial chunk to RMOD.
          AR    RADDR,R3        Adjust addr to compensate for
          S     RADDR,=F(64)      jumping into the loop.
          SRL   RMOD,1          (RMOD/4)*2 is halfword index.
          LH    RMOD,DOPEVEC9(RMOD) Use magic dope-vector for offset,
          B     LOOP(RMOD)          and jump into the loop...
*
*             Inner loop:
*
LOOP      AL    RSUM,0(,RADDR)   Add Logical fullword
          BC    12,*+6             Branch if no carry
          AR    RCARRY,RONE        Add 1 end-around
          AL    RSUM,4(,RADDR)   Add Logical fullword
          BC    12,*+6             Branch if no carry
          AR    RCARRY,RONE        Add 1 end-around
*
*                    ... 14 more replications ...
*
          A     RADDR,=F'64'    Increment address ptr
          BCT   RCOUNT,LOOP     Branch on Count
 *
 *            Add Carries into sum, and fold to 16 bits
 *
          ALR   RCARRY,RSUM      Add SUM and CARRY words
          BC    12,*+6              and take care of carry
```

```
            AR    RCARRY,RONE
            SRDL  RCARRY,16        Fold 32-bit sum into
            SRL   RSUM,16            16-bits
            ALR   RCARRY,RSUM
            C     RCARRY,=X'0000FFFF' and take care of any
            BNH   DONE                    last carry
            S     RCARRY,=X'0000FFFF'
  DONE      X     RCARRY,=X'0000FFFF' 1's complement
```

IEN 45
Section 2.4.4.5

TCP Checksum Function Design

William W. Plummer

Bolt Beranek and Newman, Inc.
50 Moulton Street
Cambridge MA 02138

5 June 1978

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

1. Introduction

Checksums are included in packets in order that errors encountered during transmission may be detected. For Internet protocols such as TCP [1,9] this is especially important because packets may have to cross wireless networks such as the Packet Radio Network [2] and Atlantic Satellite Network [3] where packets may be corrupted. Internet protocols (e.g., those for real time speech transmission) can tolerate a certain level of transmission errors and forward error correction techniques or possibly no checksum at all might be better. The focus in this paper is on checksum functions for protocols such as TCP where the required reliable delivery is achieved by retransmission.

Even if the checksum appears good on a message which has been received, the message may still contain an undetected error. The probability of this is bounded by 2**(-C) where C is the number of checksum bits. Errors can arise from hardware (and software) malfunctions as well as transmission errors. Hardware induced errors are usually manifested in certain well known ways and it is desirable to account for this in the design of the checksum function. Ideally no error of the "common hardware failure" type would go undetected.

An example of a failure that the current checksum function handles successfully is picking up a bit in the network interface (or I/O buss, memory channel, etc.). This will always render the checksum bad. For an example of how the current function is inadequate, assume that a control signal stops functioning in the network interface and the interface stores zeros in place of the real data. These "all zero" messages appear to have valid checksums. Noise on the "There's Your Bit" line of the ARPANET Interface [4] may go undetected because the extra bits input may cause the checksum to be perturbed (i.e., shifted) in the same way as the data was.

Although messages containing undetected errors will occasionally be passed to higher levels of protocol, it is likely that they will not make sense at that level. In the case of TCP most such messages will be ignored, but some could cause a connection to be aborted. Garbled data could be viewed as a problem for a layer of protocol above TCP which itself may have a checksuming scheme.

This paper is the first step in design of a new checksum function for TCP and some other Internet protocols. Several useful properties of the current function are identified. If possible

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

these should be retained in any new function. A number of plausible checksum schemes are investigated. Of these only the "product code" seems to be simple enough for consideration.

2. The Current TCP Checksum Function

The current function is oriented towards sixteen-bit machines such as the PDP-11 but can be computed easily on other machines (e.g., PDP-10). A packet is thought of as a string of 16-bit bytes and the checksum function is the one's complement sum (add with end-around carry) of those bytes. It is the one's complement of this sum which is stored in the checksum field of the TCP header. Before computing the checksum value, the sender places a zero in the checksum field of the packet. If the checksum value computed by a receiver of the packet is zero, the packet is assumed to be valid. This is a consequence of the "negative" number in the checksum field exactly cancelling the contribution of the rest of the packet.

Ignoring the difficulty of actually evaluating the checksum function for a given packet, the way of using the checksum described above is quite simple, but it assumes some properties of the checksum operator (one's complement addition, "+" in what follows):

(P1) + is commutative. Thus, the order in which the 16-bit bytes are "added" together is unimportant.

(P2) + has at least one identity element (The current function has two: +0 and -0). This allows the sender to compute the checksum function by placing a zero in the packet checksum field before computing the value.

(P3) + has an inverse. Thus, the receiver may evaluate the checksum function and expect a zero.

(P4) + is associative, allowing the checksum field to be anywhere in the packet and the 16-bit bytes to be scanned sequentially.

Mathematically, these properties of the binary operation "+" over the set of 16-bit numbers forms an Abelian group [5]. Of course, there are many Abelian groups but not all would be satisfactory for use as checksum operators. (Another operator readily

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

available in the PDP-11 instruction set that has all of these properties is exclusive-OR, but XOR is unsatisfactory for other reasons.)

Albeit imprecise, another property which must be preserved in any future checksum scheme is:

(P5) + is fast to compute on a variety of machines with limited storage requirements.

The current function is quite good in this respect. On the PDP-11 the inner loop looks like:

```
        LOOP:   ADD (R1)+,R0    ; Add the next 16-bit byte
                ADC R0          ; Make carry be end-around
                SOB R2,LOOP     ; Loop over entire packet.
```

(4 memory cycles per 16-bit byte)

On the PDP-10 properties P1-4 are exploited further and two 16-bit bytes per loop are processed:

```
LOOP: ILDB THIS,PTR   ; Get 2 16-bit bytes
      ADD SUM,THIS    ; Add into current sum
      JUMPGE SUM,CHKSU2  ; Jump if fewer than 8 carries
      LDB THIS,[POINT 20,SUM,19] ; Get left 16 and carries
      ANDI SUM,177777 ; Save just low 16 here
      ADD SUM,THIS    ; Fold in carries
CHKSU2: SOJG COUNT,LOOP ; Loop over entire packet
```

(3.1 memory cycles per 16-bit byte)

The "extra" instruction in the loops above are required to convert the two's complement ADD instruction(s) into a one's complement add by making the carries be end-around. One's complement arithmetic is better than two's complement because it is equally sensitive to errors in all bit positions. If two's complement addition were used, an even number of 1's could be dropped (or picked up) in the most significant bit channel without affecting the value of the checksum. It is just this property that makes some sort of addition preferable to a simple exclusive-OR which is frequently used but permits an even number of drops (pick ups) in any bit channel. RIM10B paper tape format used on PDP-10s [10] uses two's complement add because space for the loader program is extremely limited.

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

Another property of the current checksum scheme is:

(P6) Adding the checksum to a packet does not change the information bytes. Peterson [6] calls this a "systematic" code.

This property allows intermediate computers such as gateway machines to act on fields (i.e., the Internet Destination Address) without having to first decode the packet. Cyclical Redundancy Checks used for error correction are not systematic either. However, most applications of CRCs tend to emphasize error detection rather than correction and consequently can send the message unchanged, with the CRC check bits being appended to the end. The 24-bit CRC used by ARPANET IMPs and Very Distant Host Interfaces [4] and the ANSI standards for 800 and 6250 bits per inch magnetic tapes (described in [11]) use this mode.

Note that the operation of higher level protocols are not (by design) affected by anything that may be done by a gateway acting on possibly invalid packets. It is permissible for gateways to validate the checksum on incoming packets, but in general gateways will not know how to do this if the checksum is a protocol-specific feature.

A final property of the current checksum scheme which is actually a consequence of P1 and P4 is:

(P7) The checksum may be incrementally modified.

This property permits an intermediate gateway to add information to a packet, for instance a timestamp, and "add" an appropriate change to the checksum field of the packet. Note that the checksum will still be end-to-end since it was not fully recomputed.

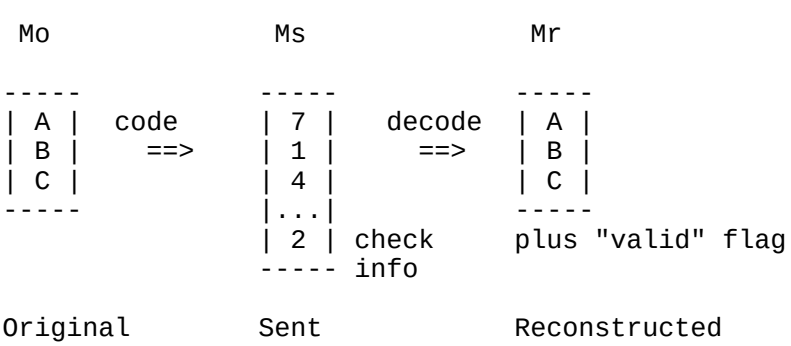
3. Product Codes

Certain "product codes" are potentially useful for checksuming purposes. The following is a brief description of product codes in the context of TCP. More general treatment can be found in Avizienis [7] and probably other more recent works.

The basic concept of this coding is that the message (packet) to be sent is formed by transforming the original source message and adding some "check" bits. By reading this and applying a (possibly different) transformation, a receiver can reconstruct

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

the original message and determine if it has been corrupted during transmission.



With product codes the transformation is Ms = K * Mo . That is, the message sent is simply the product of the original message Mo and some well known constant K . To decode, the received Ms is divided by K which will yield Mr as the quotient and 0 as the remainder if Mr is to be considered the same as Mo .

The first problem is selecting a "good" value for K, the "check factor". K must be relatively prime to the base chosen to express the message. (Example: Binary messages with K incorrectly chosen to be 8. This means that Ms looks exactly like Mo except that three zeros have been appended. The only way the message could look bad to a receiver dividing by 8 is if the error occurred in one of those three bits.)

For TCP the base R will be chosen to be 2**16. That is, every 16-bit byte (word on the PDP-11) will be considered as a digit of a big number and that number is the message. Thus,

$$Mo = \sum_{i=0 \text{ to } N} [Bi * (R^{**}i)] \quad , \quad Bi \text{ is i-th byte}$$

$$Ms = K * Mo$$

Corrupting a single digit of Ms will yield Ms' = Ms +or- C*(R**j) for some radix position j . The receiver will compute Ms'/K = Mo +or- C(R**j)/K. Since R and K are relatively prime, C*(R**j) cannot be any exact multiple of K. Therefore, the division will result in a non-zero remainder which indicates that Ms' is a corrupted version of Ms. As will be seen, a good choice for K is (R**b - 1), for some b which is the "check length" which controls the degree of detection to be had for

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

burst errors which affect a string of digits (i.e., 16-bit bytes) in the message. In fact b will be chosen to be 1, so K will be 2**16 - 1 so that arithmetic operations will be simple. This means that all bursts of 15 or fewer bits will be detected. According to [7] this choice for b results in the following expression for the fraction of undetected weight 2 errors:

f = 16(k-1)/[32(16k-3) + (6/k)] where k is the message length.

For large messages f approaches 3.125 per cent as k goes to infinity.

Multiple precision multiplication and division are normally quite complex operations, especially on small machines which typically lack even single precision multiply and divide operations. The exception to this is exactly the case being dealt with here -- the factor is 2**16 - 1 on machines with a word length of 16 bits. The reason for this is due to the following identity:

Q*(R**j) = Q, mod (R-1) 0 <= Q < R

That is, any digit Q in the selected radix (0, 1, ... R-1) multiplied by any power of the radix will have a remainder of Q when divided by the radix minus 1.

Example: In decimal R = 10. Pick Q = 6.

```
  6  =   0 * 9  +  6  =  6, mod 9
 60  =   6 * 9  +  6  =  6, mod 9
600  =  66 * 9  +  6  =  6, mod 9   etc.
```

More to the point, rem(31415/9) = rem((30000+1000+400+10+5)/9)
= (3 mod 9) + (1 mod 9) + (4 mod 9) + (1 mod 9) + (5 mod 9)
= (3+1+4+1+5) mod 9
= 14 mod 9
= 5

So, the remainder of a number divided by the radix minus one can be found by simply summing the digits of the number. Since the radix in the TCP case has been chosen to be 2**16 and the check factor is 2**16 - 1, a message can quickly be checked by summing all of the 16-bit words (on a PDP-11), with carries being end-around. If zero is the result, the message can be considered valid. Thus, checking a product coded message is exactly the same complexity as with the current TCP checksum!

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

In order to form Ms, the sender must multiply the multiple precision "number" Mo by 2**16 - 1. Or, Ms = (2**16)Mo - Mo. This is performed by shifting Mo one whole word's worth of precision and subtracting Mo. Since carries must propagate between digits, but it is only the current digit which is of interest, one's complement arithmetic is used.

```
(2**16)Mo =  Mo0 + Mo1 + Mo2 + ... + MoX +  0
    -  Mo =    - ( Mo0 + Mo1 + ......... + MoX)
---------    ---------------------------------
   Ms     =  Ms0 + Ms1 + ...              - MoX
```

A loop which implements this function on a PDP-11 might look like:

```
LOOP:   MOV -2(R2),R0   ; Next byte of (2**16)Mo
        SBC R0          ; Propagate carries from last SUB
        SUB (R2)+,R0    ; Subtract byte of  Mo
        MOV R0,(R3)+    ; Store in Ms
        SOB R1,LOOP     ; Loop over entire message
                        ; 8 memory cycles per 16-bit byte
```

Note that the coding procedure is not done in-place since it is not systematic. In general the original copy, Mo, will have to be retained by the sender for retransmission purposes and therefore must remain readable. Thus the MOV R0,(R3)+ is required which accounts for 2 of the 8 memory cycles per loop.

The coding procedure will add exactly one 16-bit word to the message since Ms < (2**16)Mo . This additional 16 bits will be at the tail of the message, but may be moved into the defined location in the TCP header immediately before transmission. The receiver will have to undo this to put Ms back into standard format before decoding the message.

The code in the receiver for fully decoding the message may be inferred by observing that any word in Ms contains the difference between two successive words of Mo minus the carries from the previous word, and the low order word contains minus the low word of Mo. So the low order (i.e., rightmost) word of Mr is just the negative of the low order byte of Ms. The next word of Mr is the next word of Ms plus the just computed word of Mr plus the carry from that previous computation.

A slight refinement of the procedure is required in order to protect against an all-zero message passing to the destination. This will appear to have a valid checksum because Ms'/K = 0/K

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

= 0 with 0 remainder. The refinement is to make the coding be Ms = K*Mo + C where C is some arbitrary, well-known constant. Adding this constant requires a second pass over the message, but this will typically be very short since it can stop as soon as carries stop propagating. Chosing C = 1 is sufficient in most cases.

The product code checksum must be evaluated in terms of the desired properties P1 - P7. It has been shown that a factor of two more machine cycles are consumed in computing or verifying a product code checksum (P5 satisfied?).

Although the code is not systematic, the checksum can be verified quickly without decoding the message. If the Internet Destination Address is located at the least significant end of the packet (where the product code computation begins) then it is possible for a gateway to decode only enough of the message to see this field without having to decode the entire message. Thus, P6 is at least partially satisfied. The algebraic properties P1 through P4 are not satisfied, but only a small amount of computation is needed to account for this -- the message needs to be reformatted as previously mentioned.

P7 is satisfied since the product code checksum can be incrementally updated to account for an added word, although the procedure is somewhat involved. Imagine that the original message has two halves, H1 and H2. Thus, Mo = H1*(R**j) + H2. The timestamp word is to be inserted between these halves to form a modified Mo' = H1*(R**(j+1)) + T*(R**j) + H2. Since K has been chosen to be R-1, the transmitted message Ms' = Mo'(R-1). Then,

```
 Ms' =  Ms*R + T(R-1)(R**j) + P2((R-1)**2)

     =  Ms*R + T*(R**(j+1))  + T*(R**j) + P2*(R**2) - 2*P2*R - P2
```

Recalling that R is 2**16, the word size on the PDP-11, multiplying by R means copying down one word in memory. So, the first term of Ms' is simply the unmodified message copied down one word. The next term is the new data T added into the Ms' being formed beginning at the (j+1)th word. The addition is fairly easy here since after adding in T all that is left is propagating the carry, and that can stop as soon as no carry is produced. The other terms can be handle similarly.

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

4. More Complicated Codes

There exists a wealth of theory on error detecting and correcting codes. Peterson [6] is an excellent reference. Most of these "CRC" schemes are designed to be implemented using a shift register with a feedback network composed of exclusive-ORs. Simulating such a logic circuit with a program would be too slow to be useful unless some programming trick is discovered.

One such trick has been proposed by Kirstein [8]. Basically, a few bits (four or eight) of the current shift register state are combined with bits from the input stream (from Mo) and the result is used as an index to a table which yields the new shift register state and, if the code is not systematic, bits for the output stream (Ms). A trial coding of an especially "good" CRC function using four-bit bytes showed showed this technique to be about four times as slow as the current checksum function. This was true for both the PDP-10 and PDP-11 machines. Of the desirable properties listed above, CRC schemes satisfy only P3 (It has an inverse.), and P6 (It is systematic.). Placement of the checksum field in the packet is critical and the CRC cannot be incrementally modified.

Although the bulk of coding theory deals with binary codes, most of the theory works if the alphabet contains q symbols, where q is a power of a prime number. For instance q taken as 2**16 should make a great deal of the theory useful on a word-by-word basis.

5. Outboard Processing

When a function such as computing an involved checksum requires extensive processing, one solution is to put that processing into an outboard processor. In this way "encode message" and "decode message" become single instructions which do not tax the main host processor. The Digital Equipment Corporation VAX/780 computer is equipped with special hardware for generating and checking CRCs [13]. In general this is not a very good solution since such a processor must be constructed for every different host machine which uses TCP messages.

It is conceivable that the gateway functions for a large host may be performed entirely in an "Internet Frontend Machine". This machine would be responsible for forwarding packets received

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

either from the network(s) or from the Internet protocol modules in the connected host, and for reassembling Internet fragments into segments and passing these to the host. Another capability of this machine would be to check the checksum so that the segments given to the host are known to be valid at the time they leave the frontend. Since computer cycles are assumed to be both inexpensive and available in the frontend, this seems reasonable.

The problem with attempting to validate checksums in the frontend is that it destroys the end-to-end character of the checksum. If anything, this is the most powerful feature of the TCP checksum! There is a way to make the host-to-frontend link be covered by the end-to-end checksum. A separate, small protocol must be developed to cover this link. After having validated an incoming packet from the network, the frontend would pass it to the host saying "here is an Internet segment for you. Call it #123". The host would save this segment, and send a copy back to the frontend saying, "Here is what you gave me as #123. Is it OK?". The frontend would then do a word-by-word comparison with the first transmission, and tell the host either "Here is #123 again", or "You did indeed receive #123 properly. Release it to the appropriate module for further processing."

The headers on the messages crossing the host-frontend link would most likely be covered by a fairly strong checksum so that information like which function is being performed and the message reference numbers are reliable. These headers would be quite short, maybe only sixteen bits, so the checksum could be quite strong. The bulk of the message would not be checksumed of course.
The reason this scheme reduces the computing burden on the host is that all that is required in order to validate the message using the end-to-end checksum is to send it back to the frontend machine. In the case of the PDP-10, this requires only 0.5 memory cycles per 16-bit byte of Internet message, and only a few processor cycles to setup the required transfers.

6. Conclusions

There is an ordering of checksum functions: first and simplest is none at all which provides no error detection or correction. Second, is sending a constant which is checked by the receiver. This also is extremely weak. Third, the exclusive-OR of the data may be sent. XOR takes the minimal amount of computer time to generate and check, but is not a good checksum. A two’s complement sum of the data is somewhat better and takes no more

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

computer time to compute. Fifth, is the one's complement sum which is what is currently used by TCP. It is slightly more expensive in terms of computer time. The next step is a product code. The product code is strongly related to one's complement sum, takes still more computer time to use, provides a bit more protection against common hardware failures, but has some objectionable properties. Next is a genuine CRC polynomial code, used for checking purposes only. This is very expensive for a program to implement. Finally, a full CRC error correcting and detecting scheme may be used.

For TCP and Internet applications the product code scheme is viable. It suffers mainly in that messages must be (at least partially) decoded by intermediate gateways in order that they can be forwarded. Should product codes not be chosen as an improved checksum, some slight modification to the existing scheme might be possible. For instance the "add and rotate" function used for paper tape by the PDP-6/10 group at the Artificial Intelligence Laboratory at M.I.T. Project MAC [12] could be useful if it can be proved that it is better than the current scheme and that it can be computed efficiently on a variety of machines.

Internet Experiment Note 45 5 June 1978
TCP Checksum Function Design William W. Plummer

References

[1] Cerf, V.G. and Kahn, Robert E., "A Protocol for Packet Network Communications," IEEE Transactions on Communications, vol. COM-22, No. 5, May 1974.

[2] Kahn, Robert E., "The Organization of Computer Resources into a Packet Radio Network", IEEE Transactions on Communications, vol. COM-25, no. 1, pp. 169-178, January 1977.

[3] Jacobs, Irwin, et al., "CPODA - A Demand Assignment Protocol for SatNet", Fifth Data Communications Symposium, September 27-9, 1977, Snowbird, Utah

[4] Bolt Beranek and Newman, Inc. "Specifications for the Interconnection of a Host and an IMP", Report 1822, January 1976 edition.

[5] Dean, Richard A., "Elements of Abstract Algebra", John Wyley and Sons, Inc., 1966

[6] Peterson, W. Wesley, "Error Correcting Codes", M.I.T. Press Cambridge MA, 4th edition, 1968.

[7] Avizienis, Algirdas, "A Study of the Effectiveness of Fault-Detecting Codes for Binary Arithmetic", Jet Propulsion Laboratory Technical Report No. 32-711, September 1, 1965.

[8] Kirstein, Peter, private communication

[9] Cerf, V. G. and Postel, Jonathan B., "Specification of Internetwork Transmission Control Program Version 3", University of Southern California Information Sciences Institute, January 1978.

[10] Digital Equipment Corporation, "PDP-10 Reference Handbook", 1970, pp. 114-5.

[11] Swanson, Robert, "Understanding Cyclic Redundancy Codes", Computer Design, November, 1975, pp. 93-99.

[12] Clements, Robert C., private communication.

[13] Conklin, Peter F., and Rodgers, David P., "Advanced Minicomputer Designed by Team Evaluation of Hardware/Software Tradeoffs", Computer Design, April 1978, pp. 136-7.