RFC 1071 : Computing the Internet checksum

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 31 mars 2013

Date de publication du RFC : Septembre 1988

https://www.bortzmeyer.org/1071.html

Un grand nombre de protocoles de la famille TCP/IP utilisent une somme de contrôle pour détecter les éventuelles corruptions de données pendant le trajet. Chacun de ces protocoles le spécifie comme il veut mais, en pratique, beaucoup utilisent le même algorithme dit « *"Internet checksum"* ». Ce RFC donne des conseils sur la mise en œuvre de cette somme de contrôle particulière.

On la trouve à beaucoup d'endroits, IPv4 (RFC 791 [1], section 3.1, « *"The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header"* », notez qu'IPv6 n'a pas de somme de contrôle), TCP (RFC 793, section 3.1, « *"The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text »), UDP (, « Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data [...]"* ») ou ICMP (RFC 4443, section 2.3, « *"The checksum is the 16-bit one's complement of the one's complement sum of the entire ICMPv6 message, starting with the ICMPv6 message type field, and prepended with a "pseudo-header" [...]"* »). Toutes ces définitions sont équivalentes : la « somme de contrôle Internet » est la somme en complément à un des seizets qui forment le pseudo-en-tête et, parfois, le message. Le pseudo-en-tête est une version simplifiée du vrai en-tête (entre autres, il ne contient pas la somme de contrôle). Par exemple, pour UDP sur IPv6, le pseudo en-tête comporte (RFC 2460, section 8.1), les adresses IP source et destination, la longueur des données, le numéro du protocole qui suit, puis l'en-tête UDP.

À l'époque où ce RFC a été écrit (il y a un quart de siècle !) les processeurs étaient bien plus lents, relativement aux réseaux, qu'aujourd'hui. Le calcul de la somme de contrôle, note notre RFC, peut donc être le facteur limitant dans un envoi de données avec TCP. C'est en partie ce qui explique que la somme de contrôle Internet soit une vraie somme, simple à calculer mais peu robuste, et pas un CRC, pourtant plus sûre (a fortiori pas une condensation cryptographique). Aujourd'hui, une mesure faite sur un PC de

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc791.txt

bureau ordinaire, en fabriquant les paquets en mode utilisateur avant de les envoyer, montre que l'envoi d'un million de courts paquets UDP sur IPv4 prend exactement le même temps avec ou sans la somme de contrôle (elle est optionnelle pour UDP sur IPv4), et `time` indique que presque tout le temps CPU a été passé dans le noyau, donc pas à calculer la somme de contrôle. (Le résultat pourrait être différent avec des paquets de plus grande taille, puisque la calcul de la somme de contrôle implique d'accéder à tout le paquet.)

Le RFC, écrit à une autre époque, estime qu'il faut chercher à optimiser vigoureusement le calcul de la somme de contrôle. Comme il est fait à chaque paquet, même un gain minime peut être intéressant.

Pour cette optimisation, le RFC note que la somme de contrôle Internet a quelques propriétés utiles (section 2) :

- Commutativité,
- Associativité,
- Indépendance par rapport à la boutianité. Que votre machine soit gros-boutienne ou petit-boutienne ne changera rien,
- Parallélisation, une conséquence de l'associativité, on peut répartir la tâche de calcul (voir le code pour le Cray plus loin).

(Il paraît que ça en fait un groupe abélien.)

Passons à l'implémentation, le cœur de ce RFC. Naturellement, la plupart des programmeurs réseaux n'auront jamais besoin de savoir calculer la somme de contrôle : quelqu'un (typiquement le noyau) le fait pour eux. Aujourd'hui, calculer la somme de contrôle est surtout utile lorsqu'on veut fabriquer soi-même la totalité du paquet, avant de l'envoyer via une prise brute <`https://www.bortzmeyer.org/raw-sockets.html`>. C'est surtout utile dans le contexte de la sécurité (faire des paquets que le système ne permettrait pas normalement de faire). Quel sont les problèmes à garder en tête ? La plupart des machines font de l'addition en complément à deux. Pour celles-ci, la solution recommandée est de faire une retenue et de l'ajouter à la fin (ce que fait le code C montré plus loin).

En IPv4, chaque routeur doit recalculer la somme de contrôle IP car le TTL change à chaque saut. On peut optimiser cette opération en faisant un recalcul incrémental (RFC 1624).

Ah, et pour vérifier la somme de contrôle ? Mêmes opérations, mais en incluant la somme de contrôle. On doit trouver uniquement des bits à Un (-0 en complément à un).

Le RFC présente en section 4 plusieurs exemples de code mettant en œuvre ces principes. D'abord, un code portable en C :

```
{
            /* 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;
    }
```

Mais il y a aussi des codes en assembleur, pour les architectures qu'on trouvait à l'époque. Par exemple, le Motorola 68020. Avec ses 20 MHz de fréquence, ce code calculait en 134 [Caractère Unicode non montré [2]]s/ko :

```
    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
```

Plus rigolo, du code assembleur pour Cray. Tirant profit des propriétés de la somme de contrôle Internet, et des caractéristiques du Cray, il effectue des calculs sur un vecteur. Même si les ordinateurs vectoriels ne sont plus à la mode, cela illustre aussi comment on peut paralléliser ce calcul :

```
      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.
```

2. Car trop difficile à faire afficher par LaTeX

Ah, et comment je fais, finalement, dans mes programmes ?

```
static          uint32_t
checksum_finish(uint32_t sum)
{
    /* Fold 32-bit sum to 16 bits */
    while (sum >> 16)
        sum = (sum & 0xffff) + (sum >> 16);
    return ~sum;
}

static          uint32_t
checksum_feed16(const void *p, unsigned int count)
{
    const uint16_t *t = p;
    uint32_t        sum = 0;
    while (count-- > 0)
        /* This is the inner checksum loop */
        sum += *t++;
    return sum;
}
```

Code qui s'utilise ainsi pour de l'UDP sur IPv6 (`buff` pointe vers les données) :

```
uint16_t
checksum6(struct ip6_hdr ip_h, struct udphdr udp_h, uint8_t * buff, uint16_t len_udp)
{
    uint32_t        sum;
    uint8_t         zero_udp_proto[] = { 0, 0, 0, SOL_UDP };
    uint32_t        length = udp_h.len;
    udp_h.check = 0;
    sum = checksum_feed16(ip_h.ip6_src.s6_addr, 8)
        + checksum_feed16(ip_h.ip6_dst.s6_addr, 8)
        + checksum_feed16(&length, 2)
        + checksum_feed16(zero_udp_proto, 2)
        + checksum_feed16(&udp_h, 4)
        + checksum_feed16(buff, len_udp / 2);
    if (len_udp % 2)
        sum += buff[len_udp - 1];
    return checksum_finish(sum);
}
```

Si vous voulez lire d'autres codes, regarder la fonction `in_cksum` dans `print-ip.c` dans le code source de tcpdump pour la vérification. Si vous regardez le code source de Linux (version 3.9), vous verrez que, pour la plupart des plate-formes que gère Linux, le code de la fonction `ip_fast_csum` est en assembleur, assembleur (en `arch/$ARCHITECTURE/include/asm/checksum.h`) que vous pouvez comparez avec les codes du RFC.

On notera que la première étude des propriétés de la somme de contrôle de l'Internet a été faite dans le très détaillé document IEN 45 <http://www.rfc-editor.org/ien/ien45.txt>, qui est reproduit à la fin de notre RFC. Ceux qui pratiquent la langue de Konrad Zuse liront avec intérêt « "*Minus Null*" <http://lutz.donnerhacke.de/Blog/Minus-Null> ». Une bonne explication sur la différence entre complément à un et complément à deux est « "*Minus Zero*" <http://www.fourmilab.ch/documents/univac/minuszero.html> ».

Merci à Lutz Donnerhacke pour ses suggestions de lecture. Merci à Kim-Minh Kaplan pour son travail sur le code.