ABPS: TED fragmentation fixing

Matteo Martelli

24 settembre 2015

1 Nelle puntate precedenti

In questo documento si fa riferimento alla modifiche apportate al componente software TED (Transmission Error Detector) riguardanti la frammentazione. Per maggiori informazioni su TED e sull'architettura Always Best Packet Switching (ABPS) si rimanda alla documentazione precedente.

Nello specifico TED offre, tramite una struttura cross-layer, un meccanismo in grado di fornire informazioni alle applicazioni riguardo l'avvenuta (o mancata) consegna al primo access point dei datagram UDP trasmessi.

Quando l'applicazione invia un datagram UDP, TED lo marca con un identificativo user-space accessibile dall'applicazione. Successivamente ogni frammento datalink spedito dall'interfaccia di rete viene associato a quell'identificativo e ad una struttura dati contenenti informazioni sullo stato del frammento. Quest'ultime vengono poi integrate con le informazioni riguardanti l'avvenuta o mancata consegna (ACK, NACK) del frammento all'access point e il relativo numero di tentativi di consegna.

Nelle versioni precedenti a questa mancava il corretto supporto per la gestione della frammentazione in IPv6. Vedremo nella prossima sezione quali modifiche sono state apportate al kernel per il corretto funzionamento con la frammentazione in IPv6. Come nella versione precedente lo sviluppo è stato continuato per il kernel 4.0.1.

Infine nella sezione 3 vedremo le modifiche apportate all'applicazione per la gestione delle notifiche di TED dei datagram frammentati oltre a qualche esempio d'utilizzo.

2 Modifiche Kernel

Prima di mostrare le modifiche effettuate nel kernel ricordiamo brevemente come sono formati gli header dei pacchetti IPv6.

2.1 Header IPv6

In particolare siamo interessati all'header del pacchetto IP che nella versione 6 è composta da un parte fissa (fixed) e una parte variabile formata da header opzionali chiamati extension header.

La figura 1 mostra il formato del fixed header IPv6.

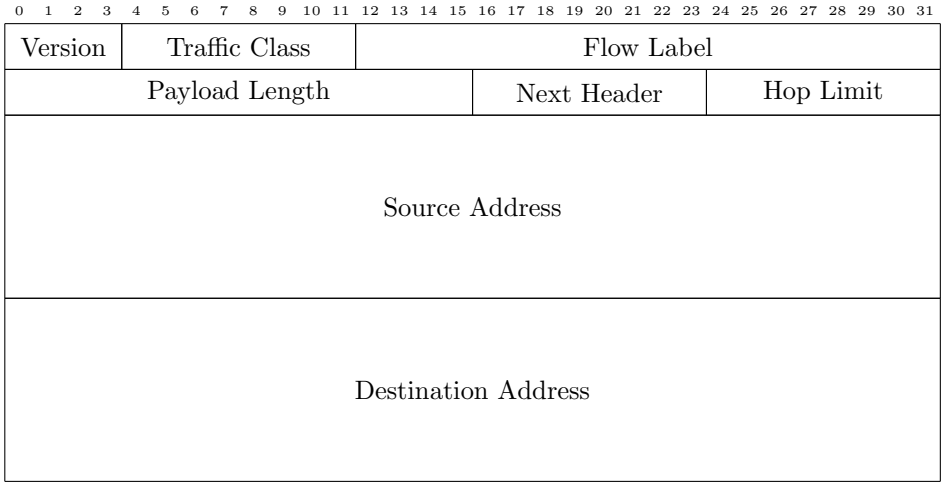

Figura 1: Formato fixed header IPv6

Notiamo che a differenza dei pacchetti IPv4 non sono presenti i campi *Flags* e *Fragment Offset* utilizzati nella versione 4 per ottenere informazioni sulla frammentazione. Per tale scopo, in IPv6, bisogna far riferimento all'extension header *Fragment*, mostrato in figura 2, nel quale sono contenute le informazioni necessarie per riasseblare i pacchetti originali.

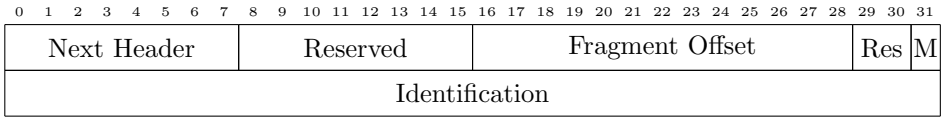

Figura 2: Formato fragment extension header IPv6. I campi *Res* e *Reserved* indicano spazio riservato inizializzato a 0. Il campo *M* si riferisce al campo *More Fragment*

Gli extension header sono situati tra il fixed header e gli header dei protocolli di livello superiore. Tramite il campo *Next Header* gli header formano una catena. In particolare il campo *Next Header* del fixed header indica il tipo del primo extension header e il *Next Header* dell'ultimo extension header (o del fixed header ove non ci fosse nessun extension header) indica il tipo dell'header del protocollo di livello superiore (ad esempio TCP o UDP).

2.2 Frammentazione IPv6 in TED

La modifica principale risiede nella funzione `ipv6_get_udp_info` la quale si occupa di recuperare le informazioni sul frammento IPv6 del relativo frame 802.11 al momento del suo invio.

Nella nuova versione si scorre innanzitutto la catena degli header fino ad incontrare l'header fragment:

```
/* Variables initialization */
*fragment_offset = *more_fragment = hdrs_len = error = 0;
nexthdr = ipv6_hdr(skb)->nexthdr;
target = NEXTHDR_FRAGMENT;

do {
        struct ipv6_opt_hdr _hdr, *hp;
        unsigned int hdrlen;
        found = (nexthdr == target);
```

```
        if ((!ipv6_ext_hdr(nexthdr)) || nexthdr == NEXTHDR_NONE) {
                break;
        }

        hp = skb_header_pointer(skb, offset, sizeof(_hdr), &_hdr);
        if (hp == NULL) {
                error = -EBADMSG;
                break;
        }

        if (nexthdr == NEXTHDR_FRAGMENT) {
                hdrlen = 8;
        } else if (nexthdr == NEXTHDR_AUTH) {
                hdrlen = (hp->hdrlen + 2) << 2;
        } else
                hdrlen = ipv6_optlen(hp);

        if (!found) {
                nexthdr = hp->nexthdr;
                offset += hdrlen;
        }

        hdrs_len += hdrlen;
} while (!found);
```

In realtà questa operazione è un controllo di sanità in quanto secondo le specifiche dell'RFC 2460[2] l'header fragment dovrebbe essere il diretto successore del fixed header. Ad ogni modo dopo aver individuato l'header fragment si può accedere alla bitmap `frag_off` della relativa struttura dati. Tramite la bitmap si possono ottenere quindi il campo *Fragment Offset* e il campo *More Fragment* del frammento:

```
/* fh is the pointer to the fragment header struct and it is retreived
according to the offset from the begging of the packet */
fh = skb_header_pointer(skb, offset, sizeof(_frag), &_frag);

if (fh) {
        *fragment_offset = ntohs(fh->frag_off) & ~0x7;
        *more_fragment = ((fh->frag_off & htons(IP6_MF)) > 0);
}
```

Infine l'ultimo campo che ci interessa ottenere è la lunghezza del frammento. La parte frammentabile è tutto quello che segue l'header fragment, quindi eventuali altri extension header IPv6, gli header dei protocolli dei livelli superiori e il frammento dati del messaggio. Bisogna quindi togliere dalla dimensione del frammento indicata dal campo *Payload* del fixed header, la dimensione dell'header fragment. Questo perchè il campo *Payload* esclude già la dimensione del fixed header e perchè l'header fragment è il diretto successore del fixed header; inoltre tutto quello che segue va considerato parte del frammento[1]:

```
*fragment_data_length = ntohs(payload_iphdr->payload_len) - hdrs_len;
```

A seguito di queste modiche il kernel con TED abilitato riesce correttamente ad ottenere le informazioni sulla frammentazione anche per IPv6.

3 Modifiche Applicazione

Al fine di verificare la corretta gestione delle notifiche TED per i datagram UDP framment, sono state apportate molteplici modifiche all'applicazione di test. Le principali verrano spiegate in questa sezione.

[1] Sempre per controllo di sanità, viene sotratto al payload tutto quello che parte dalla fine del fixed header fino alla fine dell'header fragment.

3.1 Gestione Asincrona delle Notifiche

Come prima considerazione si è dovuto modificare la gestione della ricezione delle notifiche. Nella versione precedente era stata scelta una politica sincrona, ovvero per ogni messaggio spedito si attendeva la relativa notifica TED prima di procedere alla spedizione del messaggio successivo. È evidente che tale approccio aggiunge un rallentamento notevole all'applicazione. Inoltre tale approccio non potrebbe essere applicato se l'applicazione spedisse datagram UDP di dimensione maggiore del MTU inquanto per ogni datagram spedito bisognerebbe attendere le notifiche di tutti i frammenti (ricordiamo che TED invia una notifica per ogni frame 802.11 spedito, quindi per ogni frammento del datagram UDP originale).

Si è scelto quindi di passare ad una gestione asincrona delle notifiche. Esistono diversi articoli[3, 1] che mostrano e comparano i possibili sistemi per gestire in modo asincrono i socket descriptor. Il recente *epoll* introduce varie ottimizzazioni in termini prestazionali rispetto a *select* e *poll*. Tuttavia tali ottimizzazioni sembrerebbero avere effetto solo con un gran numero di socket e al contrario sembrano peggiore le prestazioni con un numero basso di socket descriptor.

È da considerare inoltre che l'applicazione di test è un primo passo verso l'implementazione di un proxy client descritto da ABPS[4] in cui il numero di socket da gestire è piccolo: generalmente un socket per l'applicazione, e due o tre per il numero di interfacce.

Ad ogni modo è molto interessate la gestione degli eventi in modalità edge-triggered introdotto da epoll. Nella modalità edge-triggered gli eventi vengono segnalati all'applicazione solo c'è un cambio di stato nei file descriptor monitorati. Vediamo di seguito un esempio dal man di epoll per chiarezza.

> Suppose that this scenario happens:
>
> 1. The file descriptor that represents the read side of a pipe (rfd) is registered on the epoll instance.
> 2. A pipe writer writes 2 kB of data on the write side of the pipe.
> 3. A call to epoll_wait(2) is done that will return rfd as a ready file descriptor.
> 4. The pipe reader reads 1 kB of data from rfd.
> 5. A call to epoll_wait(2) is done.
>
> If the rfd file descriptor has been added to the epoll interface using the EPOLLET (edge-triggered) flag, the call to epoll_wait(2) done in step 5 will probably hang despite the available data still present in the file input buffer; meanwhile the remote peer might be expecting a response based on the data it already sent.
>
> ...
>
> The suggested way to use epoll as an edge-triggered (EPOLLET) interface is as follows:
>
> i with nonblocking file descriptors; and
>
> ii by waiting for an event only after read(2) or write(2) return EAGAIN.

Nel caso in cui la nostra applicazione inviasse datagram UDP molto grandi (supponiamo $N * MTU$) verso una socket *sd*, per ogni messaggio inviato avremmo N notifiche TED provenienti dal kernel per *sd*. Quindi con invii di messaggi molto frequenti da parte dell'applicazione, si potrebbe avere un gran overhead di segnalazione eventi da parte del kernel se venisse effettuata per ogni N frammenti del messaggio originale. Utilizzando l'edge-triggered invece avremmo un segnale solo al primo messaggio TED introdotto nella coda degli errori (ERRQUEUE). I successivi messaggi TED verrano introdotti, senza essere segnalati all'applicazione, nella ERRQUEUE che dovrà però venir svuotata ogni volta che risulta essere non vuota.

Di seguito vediamo il main loop dell'applicazione di test:

```
/* Main epoll loop. Wait for events on the socket descriptor.
 * If an EPOLLOUT event is triggered a new message can be sent.
 * If an EPOLLERR event is triggered some ted error message
 * may be present in the errqueue. */
for (;;) {
```

```
        nfds = epoll_wait(epollfd, events, MAX_EPOLL_EVENTS, -1);
        if (nfds == -1)
                utils_exit_error("epoll_wait error\n");

        for (i = 0; i < nfds; i++) {

                /* Send a new message if the socket is ready for writing */
                if (events[i].events & EPOLLOUT && idx < conf.n_packets) {
                        send_new_msg(hashtb);
                        idx++;
                }

                /* If there are pending TED error messages
                 * in the errqueue, receive them and print TED infos. */
                if (events[i].events & EPOLLERR)
                        recv_ted_errors(hashtb);

        }
}
```

Essenzialmente vengono inviati un numero di pacchetti `conf.n_packets` (di default o specificati dall'utente) ad un server e si controlla dopo ogni invio se ci sono messaggi di errore, tramite l'evento EPOLLERR, per il socket descriptor monitorato (in questo caso ne è solamente uno). Se ci sono messaggi di errore pendenti viene invocata la funzione `recv_ted_errors()` che essenzialemente legge le notifiche TED dalla coda ERRQUEUE fino a che non risulta vuota (recv ritorna EAGAIN). Per ogni notifica ricevuta l'applicazione riempie una struttura `ted_info_s` che verrà utilizzata per ricomporre i frammenti e/o fornire informazioni sullo stato del pacchetto originale:

```
struct ted_info_s {
        uint32_t msg_id;
        uint8_t retry_count;
        uint8_t status;
        uint16_t frag_length;
        uint16_t frag_offset;
        uint8_t more_frag;
        uint8_t ip_vers;
        char *msg_pload;
};
```

3.2 Gestione della Frammentazione

Vediamo adesso come vengono gestite le notifiche TED dei messaggi frammentati. Ricordiamo che per ogni messaggio UDP abbastanza grande da venir frammentato, vengono ricevute un numero di notifiche TED pari al numero di frammenti del messaggio originale. Lo scopo dell'applicazione di test è quello di riassemblare le notifiche frammentate in modo da fornire delle informazioni medie sul corrispondente messaggio precedentemente inviato.

A tal proposito viene utilizzata un hashtable di strutture riguardanti i messaggi inviati:

```
struct msg_info_s {
        int size;
        uint32_t id;
        struct ted_info_s *frags[MAX_FRAGS];
        int n_frags;
        short int last_frag_received;
};
```

Quindi di ogni messaggio inviato viene salvata la dimensione `size` e l'identificatore `id` assegnato dal kernel. Inoltre vengono predisposte le variabili utilizzate in seguito per ricomporre i frammenti: `frags`

è un array di struct `ted_info_s` le quali andranno a contenere le informazioni di ogni singola notifica TED; `n_frags` è il numero di frammenti attualmente ricevuti (inizializzato a 0) e `last_frag_received` è un booleano che indica se è stato ricevuto o meno l'ultimo frammento (per posizione)[2].

Oltre a memorizzare nell'hashtable ogni messaggio spedito, per ogni notifica TED ricevuta viene controllato se quest'ultima non è una notifica di un frammento (oppure lo è), ovvero se i valori *morefrag* e *frag_offset* risultano entrambi zero (o altrimenti). Nel caso in cui fosse un frammento viene cercata nell'hashtable il messaggio originale corrispondente in base all'identificatore `msg_id` presente nella struct `ted_info_s` della notifica. Ricordiamo infatti che TED assegna ad ogni notifica di un frammento l'id del messaggio originale.

Una volta riottenuta dall'hashtable la struttura `msg_info_s` del meessaggio originale vi si aggiunge la struct `ted_info_s` della notifica all'array `frags`.

Questa operazione viene effettuata per ogni notifica TED di un frammento ricevuta, finchè non vengono ricevuti tutti i messaggi. Se viene ricevuta la notifica del frammento che ha valore `more_frag` uguale a 0, ovvero l'ultimo frammento di posizione, oppure se viene ricevuta una notifica di qualsiasi frammento ma è già stato ricevuto l'ultimo di posizione, si procede ad ordinare, secondo gli offset, tutti i frammenti già ricevuti. Se la somma delle dimensioni di ciascun frammento è uguale alla dimensione del messaggio originale (più l'header UDP) allora si considerano validi i frammenti e si calcolano la media del numero di *ack* e di *retry count* del messaggio originale. Quest'ultima operazione viene eseguita dalla funzione `try_recompose` riportata qui di seguito:

```
int try_recompose(struct msg_info_s *msg_origin)
{
        ...
        /* Sort fragments by the fragment offsets */
        qsort(msg_origin->frags, msg_origin->n_frags,
              sizeof(struct ted_info_s *), compare_frags);

        tot_len = 0;
        acked_avg = retry_count_avg = 0.0;
        if (msg_origin->frags[msg_origin->n_frags - 1]->more_frag == 0) {
                for (i = 0; i < msg_origin->n_frags; i++) {
                        tot_len += msg_origin->frags[i]->frag_length;
                        acked_avg += (float)msg_origin->frags[i]->status;
                        retry_count_avg += (float)msg_origin->frags[i]->retry_count;
                }
                acked_avg /= (float)msg_origin->n_frags;
                retry_count_avg /= (float)msg_origin->n_frags;
        }
        if (tot_len != msg_origin->size + UDP_HEADER_SIZE) {
                printf("recomposition failed.\n");
                return 0;
        }
        return 1; /* Success */

}
```

Non c'è ancora una gestione dell'errore: potrebbe accadere che arrivino le notifiche di tutti i frammenti ma la somma delle loro dimensioni non corrisponda al messaggio originale. Inoltre potrebbe accadere che non tutte le notifiche arrivino. Un'idea di sviluppo potrebbe essere quella di controllare periodicamente l'hashtable e rimuovere quei messaggi che non hanno ricevuto tutte le notifiche entro un certo tempo limite. Tali considerazioni potrebbero essere rianalizzate per le future versioni.

[2]Non necessariamente l'ultimo frammento per posizione viene ricevuto cronologicamente dopo i suoi predecessori

3.3 Refactoring e Organizzazione

Rispetto alla versione precedente è stata effettuata una consistente riorganizzazione e un refactoring del codice sorgente. Innanzitutto sono state unificate le due applicazioni per IPv4 e IPv6 in un'unica applicazione potendo scegliere all'invocazione quale versione utilizzare.

Inoltre la gestione dei log in formato json presente nella versione precedente è stata rimossa in quanto non più applicabile con la gestione asincrona delle notifiche TED.

Infine è stata riorganizzata la struttura file portando tutte le constanti e le strutture in file header appositi, `consts.h` e `structs.h`, aggiungendo i sorgenti per funzioni di utilità generiche, `utils.c/h`, e separando le funzioni di rete dalle funzioni di gestione delle notifiche TED relativamente in `network.c/h` e `tederror.c/h`.

3.4 Utilizzo

Riferimenti bibliografici

[1] *select / poll / epoll: practical difference for system architects*, 2014. `http://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects`.

[2] Stephen E Deering. Internet protocol, version 6 (ipv6) specification. 1998.

[3] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Linux Symposium*, volume 1, 2004.

[4] Vittorio Ghini, Giorgia Lodi, and Fabio Panzieri. Always best packet switching: the mobile voip case study. *Journal of communications*, 4(9):700–713, 2009.