Capitolo 1

Transmission Error Detector

In questo capitolo si vogliono illustrare le fasi di progettazione e implementazione di un modulo TED per ABPS su kernel Linux 4.0.
Nel precedente capito si è descritto brevemente il funzionamento di TED e il suo ruolo nel sistema ABPS ideato come supporto alla mobilità.
Il meccanismo di Transmission Error Detection può essere applicato a qualsiasi interfaccia di rete di un dispositivo mobile.
In questa capitolo verrà illustrata la progettazione di un modulo TED per Wi-Fi.

1.1 Design and implementation

Transmission Error Detector è implementato in maniera *cross-layer* lungo lo stack di rete del kernel Linux e ha lo scopo di monitorare ciascun datagram UDP in invio da una certa interfaccia di rete e notificare ad un eventuale ABPS proxy client lo stato di consegna del pacchetto all'access point in modo tale da poter valutare la QoS del collegamento attualmente utilizzato per la trasmissione.
Per realizzare ciò TED introduce una notifica di tipo *First-hop Transmission Notification* che sarà fatta pervenire al proxy client.
In particolare in fase di invio di un pacchetto questo verrà marcato con

un particolare identificativo che sarà fatto pervenire all'applicativo mittente. Quando il messaggio starà per essere spedito dall' interfaccia di rete Wi-Fi il sottosistema mac80211 assegnerà al frame un sequence number tramite uno dei suoi handler: il sequence number assegnato al frame sarà associato con l'identificativo assegnato al messaggio in precedenza da TED e mantenuto in una struttura dati creata ad-hoc.
Una volta che il frame sarà stato effettivamente spedito e il sottosistema mac80211 avrà ricevuto lo status di consegna di un dato frame al first-hop si andrà alla ricerca nell'apposita struttura dati dell'identificativo associato al sequence number del frame di cui si è appena scoperto lo stato di consenga: l'elemento verrà rimosso dalla struttura dati e verrà sollevata la notifica First-hop Transmission Notification all'applicazione interessata a monitorare il messaggio con quello specifico identificativo indicando lo status di consegna.
Quest'approccio è stato adottato in qualsiasi funzione che interagisce con la rete tramite interfaccia socket non è bloccante fino all'invio effettivo del messaggio ma solamente fino a quando tutti i socket buffer sono stati alloccati è posti in apposite code di trasmissione dello stack di rete.
Qui di seguito verrà descritto nel dettaglio TED e la sua implementazione sviluppata lungo tutto lo stack di rete Linux.

1.1.1 Application layer

Le applicazioni per le quali ABPS vuole essere un supporto alla mobilità sono le applicazioni multimedia-oriented che, come già trattato nei capitoli precedenti, sono solitamente progettate sopra UDP. Un'applicazione che utilizza UDP per le sue comunicazioni di rete sfrutta uno o più socket datagram ovvero di tipo connectionless.
Per poter valutare la ritrasmissione di un dato messaggio il proxy client necessita di un meccanismo di identificazione per ciascun datagram. A tal proposito si è esteso la system call *sendmsg* in modo tale che possa ritornare un **id** univoco per il messaggio in invio cosicché il proxy client possa mante-

nerlo e usarlo per riferirsi a quel preciso datagram. Tutte le notifiche ricevute poi in seguito dal proxy client e provenienti da TED faranno riferimento a un datagram utilizzando lo stesso identificativo.
La system call sendmsg consente di inviare, assieme al contenuto del messaggio, delle informazioni di controllo aggiuntive dette *ancillary data* che non saranno però trasmesse lungo la rete. Dal punto di vista implementativo i dati di tipo ancillary sono realizzati in POSIX tramite una sequenza di strutture **struct cmsghdr** contenenti le informazioni di controllo. L'estensione di sendmsg, progettata per realizzare TED, prevede l'utilizzo di ancillary data durante la fase di invio di un messaggio; in particolare:

- Per segnalare a TED che l'applicativo invocante la system call sendmsg richiede di poter ricevere l'id per il messaggio in invio. Per far ciò viene introdotto un nuovo valore non utilizzato per il campo **cmsg_type** della struct cmsghdr.

- Per passare a TED un indirizzo di memoria in user space dove TED potrà assegnare l'identificativo generato per il datagram in invio.

Una volta creato un socket può essere possibile, inoltre, specificare particolari opzioni aggiuntive da adottare a un certo livello di rete e protocollo tramite la system call *setsockopt*. Con questo meccanismo è possibile abilitare un socket per la ricezione di messaggi di tipo ICMP (specificando come parametro della system call setsockopt l'opzione IP_RECVERR): ad esempio quando avviene un errore di trasmissione su uno dei nodi intermedi (ad esempio un router non riesce a determinare la destinazione di un certo datagram) viene generato un messaggio di tipo ICMP e consegnato all'host mittente che sarà quindi accodato nel buffer del socket.
I messaggi di errore possono essere poi letti dall'applicazione che mantiene il socket tramite la system call recvmsg specificando il socket ed il flag MSG_ERRQUEUE.
In TED questo meccanismo è stato sfruttato per la ricezione delle notifiche di tipo First-hop Transmission Notification.

In particolare in Linux eventuali messaggi di errore che possono provenire dalla rete sono mantenuti in una struttura di tipo struct sock_extended_err; uno dei campi di questa struttura è il campo ee_type che definisce il tipo di messaggio a cui si fa riferimento. È stato, quindi, aggiunto un nuovo valore (SO_EE_ORIGIN_LOCAL_NOTIFY) tra quelli disponibili per il campo ee_type indicante la notifica proveniente da TED.
Maggiori dettagli su come la notifica viene effettivamente generata e di come la struttura struct sock_extended_err sia stata utilizzata (ed estesa) per adattarla al meglio al nostro obbiettivo verranno dati in seguito.
Ricapitolando, quindi, gli unici accorgimenti che un applicativo deve adottare per beneficiare dei meccanismi forniti dalla versione di Transmission Error Detector sviluppata consistono in:

- Adottare la system call sendmsg e specificare nell'ancillary data del messaggio in invio che si è interessati a ricevere un identificativo per quel messaggio oltre che all'indirizzo di memoria dove si vuole venga memorizzato l'identificativo assegnato.

- Abilitare il socket UDP utilizzato per la trasmissione alla ricezione degli errori.

1.1.2 Transport layer

Una volta che il messaggio sarà stato inviato tramite sendmsg ed il Socket Interface Layer avrà passato il controllo alla primitiva di livello trasporto udp_sendmsg, tramite una funzione appositamente sviluppata, si andrà ad analizzare i dati di tipo ancillary contenuti nel messaggio passato come parametro della udp_sendmsg. La struttura struct msghdr mantiene il messaggio passato alla system call dall'applicativo a livello utente e gli eventuali dati di controllo: una volta analizzati gli ancillary data se l'applicazione ha specificato di essere interessata a ricevere l'identificativo del messaggio inviato verrà settato un apposito flag is_identifier_required e verrà mantenuto l'indirizzo di memoria user space specificato dall'applicativo dove TED potrà

settare l'identificativo calcolato per il pacchetto in invio.
Quando il controllo viene passato momentaneamente a livello rete, come descritto nel capitolo 2, e verrà allocato il socket buffer relativo al messaggio in invio sarà calcolato l'identificativo per quel pacchetto e assegnato alla struttura sk_buff (maggiore dettagli in seguito).
Non appena il flusso dell'esecuzione viene ripreso dalla primitiva udp_sendmsg, se l'allocazione del socket buffer è andata a buon fine, nell'indirizzo di memoria user space, precedentemente ricavato dall'ancillary data del messaggio, verrà copiato l'identificativo appena assegnato alla struttura sk_buff tramite primitiva put_user (macro che consente di copiare un certo valore presente in kernel space a partire da un certo indirizzo user space).

1.1.3 Network Layer

A livello rete sono state implementate alcune modifiche in particolare ogni qualvolta viene allocata una struttura di tipo sk_buff per un messaggio in invio verrà calcolato un identificativo univoco per quel pacchetto e assegnato alla struttura stessa.
Nella precedente versione di TED, sviluppata per Kernel Linux 2.6.30-rc5, veniva utilizzato come identificativo da utilizzare a livello applicativo per monitorare i singoli messaggi spediti il campo ID (per maggiori dettagli si veda il primo capitolo) del datagram IPv4: una volta che veniva calcolato e settato questo campo della struttura iphdr, che rappresenta un header IPv4 nei moduli di rete del kernel, il valore veniva passato a livello utente.
Nella versione sviluppata per kernel Linux 4.0 viene introdotto il supporto a IPv6.
L'header IPv6, supportando solamente la frammentazione end-to-end, non è caratterizzato dal campo Identification come l'header della precedente versione dell'Internet Protocol.

Per ovviare a questo problema si è definito un contatore globale: ogni volta che viene allocato un nuovo socket buffer il contatore viene incrementato (

all'interno di una critical section) e associato a quel socket buffer.
La struttura sk_buff è stata quindi estesa aggiungendo il nuovo campo che manterrà l'identificativo univoco assegnatoli da TED.

```
struct sk_buff
{
        ...
        ...
        uint32_t sk_buff_identifier;
};
```

È stata aggiunta, inoltre, una funzione per l'assegnazione dell'identificativo al socket buffer.

Quando la funzione di livello trasporto udp_sendmsg lascia il controllo alla primitiva __ip_append_data di livello rete per ogni sk_buff allocato verrà inizializzato il suo identificativo tramite la nuova primitiva introdotta.
Non appena la funzione udp_sendmsg avrà ripreso il controllo dell'esecuzione potrà così notificare, all'applicativo che ne ha fatto richiesta, l'identificativo appena assegnato al messaggio prima di continuare con il workflow di trasmissione dei messaggi.
A livello rete può essere necessario frammentare un pacchetto prima di inviarlo: dal punto di vista implementativo questo si traduce nell'allocare un socket buffer per ogni frammento di un datagram IP. Questo viene realizzato dal modulo di rete all'interno della routine ip_fragment. Il meccanismo è stato esteso copiando all'interno del nuovo socket buffer allocato il valore dell'identificativo mantenuto dal datagramma originario: in questo modo ciascun fragment di uno stesso datagram IP avrà lo stesso identificativo.
Il meccanismo precedentemente descritto è stato equivalentemente realizzato sia per il protocollo IPv4 che IPv6.

1.1.4 The mac80211 subsystem

Quando un pacchetto a livello data-link deve essere spedito tramite interfaccia di rete wireless, come già descritto nel capitolo 2, il controllo viene lasciato al modulo mac80211 a cui è delegato il compito di generare, trasmettere e ricevere frames 802.11.
In fase di preparazione dell'header IEEE 802.11 verrà invocato, tra gli altri, un apposito handler chiamato ieee80211_tx_h_sequence che si occupa di generare e assegnare all'header 802.11 il sequence number.
In fase di invio di un frame 802.11 i dati in trasmissione sono wrappati all'interno di una struttura ieee80211_tx_data che mantiene un riferimento a una struttura socket buffer. Una volta assegnato il sequence number all'header del frame in uscita sarà possibile accedere alla struttura sk_buff (a cui i dati in trasmissione fanno riferimento) e quindi al socket associato.
TED può verificare se su quel socket è stata abilitata l'opzione di ricezione di eventuali messaggi d'errore o meno. Se l'opzione è stata attivata in precedenza a livello applicativo (tramite system call setsockopt) il sequence number calcolato viene associato all'identificativo sk_buff_identifier del socket buffer di cui si sta preparando l'header 802.11; l'identificativo del socket buffer e il sequence number saranno memorizzati assieme all'interno di una struttura dati ad-hoc mantenuta da TED.
In questra struttura vengono mantenute, per ogni messaggio, anche altre informazioni oltre al sequence number e l'identificativo assegnato da TED.
In particolare viene memorizzato se il messaggio è incapsulato in un datagram IPv4 o IPv6. Ciò è possibile verificando il campo Protocol ID mantenuto all'interno dell'header IEEE 802.2 (per maggiori spiegazioni si veda il capitolo 1). A livello implementativo TED accede a questo campo tramite un offset a partire dall'header IEEE 802.11.
Se il contenuto del messaggio è IPv4 viene anche memorizzato se vi sono altri frammenti dello stesso datagram IP, la lunghezza del frammento e il fragment offset.

Come già accennato nel capitolo 2 una volta che un frame IEEE 802.11 è stato inviato attraverso un'interfaccia di rete wireless lo stato di trasmissione sarà notificato in maniera asincrona tramite uno specifico handler chiamato ieee80211_tx_status che fornisce alcune informazioni sulla trasmissione avvenuta. Alcune delle informazioni fornite da questo handler saranno poi quelle utilizzate per il contenuto della notifica di tipo First-hop Transmission Notification che TED solleverà verso il proxy client ABPS.
In particolare l'handler ieee80211_tx_status è stato esteso in modo tale da estrarre dall'header 802.11 associato al messaggio di cui l'handler sta notificando lo status e da questo salvare il sequence number: il sequence number sarà utilizzato come chiave di ricerca all'interno della struttura di dati, precedentemente menzionata, per trovare un eventuale identificativo associato a quel sequence number. Se la ricerca si conclude con successo l'elemento viene rimosso dalla struttura dati e TED si preparerà ad inviare una notifica all'applicazione che sta monitorando il datagram con l'identificativo risultante.

1.1.5 First-hop Transmission Notification

A seconda se la trasmissione dati sfrutta IPv4 o IPv6 la notifica verso il proxy client ABPS o più genericamente verso un'applicazione che sfrutta TED sarà strutturata in modo diverso. Come già brevemente accennato la notifica di tipo First-hop Transmission Notification viene realizzata estendendo la struttura di tipo struct sock_extended_err tradizionalmente utilizzata per identificare errori.

```
struct sock_extended_err
 {
        __u32   ee_errno;
        __u8    ee_origin;
        __u8    ee_type;
        __u8    ee_code;
        __u8    ee_pad;
```

```
        __u32   ee_info;
        __u32   ee_data;

        /* new value added for retry count */
        __u8    ee_retry_count;
};

#define SO_EE_ORIGIN_NONE       0
#define SO_EE_ORIGIN_LOCAL      1
#define SO_EE_ORIGIN_ICMP       2
#define SO_EE_ORIGIN_ICMP6      3
#define SO_EE_ORIGIN_TXSTATUS   4

/* new value for ee_type */
#define SO_EE_ORIGIN_LOCAL_NOTIFY 5
```