

# 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 consegna: l'elemento verrà rimosso dalla struttura dati e verrà sollevata la notifica First-hop Transmission Notification all'app interessata a monitorare il messaggio con quello specifico identificativo indicando lo status di consegna.

Quest'approccio è stato adottato in quanto la `sendmsg` da noi utilizzata (ma lo stesso comportamento è adottato da 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 allocati e posti in appositi buffer del workflow di rete.

Qui di seguito verrà descritto nel dettaglio TED e la sua implementazione 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` in particolare prevede l'utilizzo di ancillary data in fase di invio:

- 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 `u_int8_t 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 obiettivo 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 funzione di livello trasporto `udp_sendmsg`, come spiegato nel capitolo 2, tramite una funzione appositamente sviluppata e contenuta nel file `socket.c` del kernel Linux, si andrà ad analizzare i dati di tipo ancillary contenuti nella struttura `struct msghdr` passata 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 l'app dove TED po-

trà 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 dagli ancillari data del messaggio passato a livello udp tramite primitiva `put_user` ( macro che consente di copiare un certo valore presente in kernel space a partire da un certo indirizzo user space ) verrà copiato l'identificativo.

### 1.1.3 Network Layer

A livello rete sono state implementate alcune modifiche in particolare verrà calcolato l'identificativo di ciascun pacchetto e assegnato a una struttura di tipo `sk_buff` appena viene allocata.

Nella precedente versione sviluppata di TED per Kernel Linux 2.6.30-rc5 come identificativo da utilizzare a livello applicativo per monitorare i singoli messaggi spediti veniva usato 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, 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 al 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 (

in 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 maggiori dettagli si veda il capitolo 2 ) 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 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 verrà lasciato al modulo mac80211 a cui è delegato il compito di generare trasmettere e ricevere frame 802.11.

In fase di preparazione dell'header IEEE 802.11 verrà invocato un apposito handler chiamato `mac80211_tx_h_sequence` verrà