22.4.'15
Gabriele Di Bernardo
0000634927

Oggetto: TED per ABPS su kernel Linux 3.6 installazione e tests

Abstract

In questo documento si riportano i risultati di alcuni test di TED per ABPS su kernel Linux v3.6.

TED, Transmission Error Detector, può essere pensato come ad un modulo che estende il WiFi introducendo un meccanismo cross-layer su più livelli dello stack di rete: il suo compito è quello di notificare a un eventuale client ABPS o ad un'applicazione in genere lo stato di consegna (se consegnato o meno) di un messaggio ad un access point (first hop).

Implementazione

L'implementazione di TED su kernel Linux avviene appunto su più livelli dello stack di rete.

Socket buffer

Una struttura dati largamente utilizzata su tutto lo stack di rete del kernel è sk_buff. La struttura sk_buff mantiene tutte le informazioni relative a un pacchetto in invio o ricezione su tutti i livelli dello stack: ovvero tutti i livelli dello stack di rete coinvolti nell'invio o ricezione di un pacchetto utilizzano la struttura sk_buff andando a operare su opportuni campi della struttura stessa (headers).
La struttura dati inoltre contiene puntatori utili alla realizzazione di una lista circolare.

Dall'app al device driver

Quando un'applicazione vuole inviare un pacchetto dati di tipo UDP si appoggia su un'interfaccia di tipo socket e vengono quindi messe a disposizione una serie di primitive e funzioni (come ad esempio sendmsg).
La funzione sendmsg consente di inviare, assieme ad un dato messaggio, dati di tipo ancillary contenenti ulteriori informazioni di controllo.
Una volta invocata una funzione del socket interface layer il controllo viene passato al kernel ed in particolare al livello trasporto: quindi supponendo di inviare un pacchetto UDP tramite sendmsg l'esecuzione viene ripresa dalla primitiva udp_sendmsg che esegue alcuni controlli per poi dare momentaneamente controllo alla funzione di livello rete ip_append_data che suddivide lo sk_buff originario in tanti buffer più piccoli a seconda del valore della MTU per poi accodarli nella coda in uscita del socket.
Dopodichè il controllo viene passato a udp_push_pending_frames che si occupa di generare l'header UDP per poi passare definitivamente il controllo al livello network.

Nella funzione di livello rete ip_push_pending_frames viene preparato l'header IP per ogni pacchetto in coda sul socket e dopo qualche controllo vengono invocate le funzioni utili per il routing che sposteranno poi il flusso dell'esecuzione a livello data link.

Il livello data link nel caso dell'invio di un pacchetto attraverso scheda di rete wireless si appoggia sul modulo 802.11 che ha come entry point la funzione ieee80211_xmit che si occupa di calcolare l'header 802.11 e di eseguire una serie di operazione tramite degli handler opportunamente settati, ad esempio ieee80211_tx_h _sequence calcola il sequence number del frame 802.11.
Il frame in uscita viene quindi passato al device driver opportuno per l'invio.
Il buffer relativo al messaggio in uscita non viene però immediatamente eliminato in quanto se non viene ricevuto un ACK per il frame in uscita il modulo 802.11 riprova a re-inviare il pacchetto per un certo numero di volte.
Alla fine di questo processo viene invocata la routine ieee80211_tx_status che fornisce l'esito della trasmissione per un pacchetto (se è stato effettivamente consegnato al fist hop) ed ulteriori informazioni come ad esempio quante volte il pacchetto è stato ritrasmesso attraverso l'interfaccia di rete (retry_count).

Implementazione Transmission Error Detector

Il compito principale del TED è quello di notificare ad un'applicazione client lo stato di un dato messaggio inviato.

Per far ciò la coda degli errori è stata estesa con un nuovo messaggio di tipo SO_EE_ORIGIN_LOCAL_NOTIFY.
Un'applicazione che si interfaccia sulla rete tramite socket UDP, attraverso la system call setsockopt, può configurare una serie di parametri e opzioni per il socket stesso tra le quali è possibile configurare il socket per la ricezione di messaggi in caso di errore: ad esempio quando avviene un errore in trasmissione viene generato un messaggio di tipo ICMP che sarà accodato sul buffer del socket. L'applicazione potrà quindi (tramite recvmsg specificando flag MSG_ERRQUEUE) leggere i messaggi d'errore in coda sul buffer.

Un'applicazione che riceve delle notifiche riguardo lo status di uno dei suoi messaggi inviati dovrà però poter distinguere e disambiguare le notifiche in base al messaggio alla quale esse fanno riferimento. Per far ciò ciascun pacchetto sarà marcato con un identificativo univoco.
La versione di TED per ABPS realizzata su kernel Linux 2.6.30-rc5 utilizza come identificativo del pacchetto il campo Identification dell'header IPv4. La versione realizzata per kernel v3.6, che introduce il supporto a IPv6, sfrutta un unico contatore incrementato per ogni pacchetto in uscita (sia per IPv4 che IPv6) questo perché il campo id nell'header IPv6 è stato rimosso.

Quando viene inviato un messaggio UDP tramite sendmsg il client app provvederà ad inserire nell'ancillary data un puntatore a un'area di memoria user space. Quando il kernel comincerà a processare l'invio di quel dato messaggio controllerà l'ancillary data del pacchetto, prenderà il puntatore all'area di memoria in esso contenuto e lo userà per passare l'identificativo per quel messaggio

all’applicazione (sendmsg termina prima dell’effettivo inoltro del messaggio sulla scheda di rete). Tutti gli eventuali fragments del messaggio originario ovviamente avranno lo stesso identificativo (l’id verrà copiato in ogni frammento in ip_fragment, funzione che si occupa di frammentare pacchetto a livello network).

Quando il messaggio in uscita arriva al modulo 802.11 questo passa attraverso alcuni handler prima dell’effettiva consegna alla scheda di rete. Uno di questi handler, ieee80211_tx_h_sequence, assegnerà al frame in uscita un certo sequence number: a questo punto il messaggio originario assieme al sequence number verranno memorizzati in una lista ad-hoc (se il socket relativo è stato inizializzato e opportunamente configurato per la ricezione di notifiche).

Quando il sistema avrà ricevuto un responso per un dato frame 802.11 inviato e se il frame contiene un pacchetto UDP tramite il sequence number si andrà alla ricerca di quel dato messaggio all’interno della lista mantenuta ad-hoc: l’elemento verrà rimosso dalla lista e verrà notificato lo status del messaggio all’app relativa inviando un messaggio di local notify su quel socket.

Installazione custom kernel & test

Installazione Kernel v3.6

Ho eseguito dei test sulla versione di TED per ABPS sviluppata per kernel Linux v3.6.
Per far ciò come prima cosa ho scaricato dalla repository ufficiale la versione 3.6 del kernel Linux.

Una volta scaricato il kernel, tramite gli scripts forniti assieme al codice sorgente di TED, ho integrato i moduli modificati al kernel originario.

Per la compilazione del kernel ho usato il Makefile.

Per prima cosa ho generato una configurazione per il kernel (ho usato una configurazione di default) tramite il comando make menuconfig che fornisce una

piccola interfaccia di testo con tutte le opzioni di configurazione (per eseguire questo comando è necessario aver installato libncurses e libncurses-devel).

Dopodichè ho eseguito i seguenti comandi:

`# make`
Per la compilazione del main kernel

`# make modules`
Per compilare i diversi moduli

`# make modules_install`
Per installare moduli compilati

`# make install`
Per installare il kernel nel sistema

Per utilizzare il nuovo kernel appena compilato sarà sufficiente riavviare la macchina.

Test
Per testare il sistema ho sfruttato i file forniti nella directory testing del progetto ABPS per kernel v3.6. Si tratta di due semplici applicativi scritti in Python:
- recv.py, crea un socket UDP IPv6 e si mette in ascolto di eventuali messaggi;
- ctypestest.py, invia dei pacchetti ad un socket UDP IPv6 (sfruttando una libreria scritta in C dove all'interno viene utilizzata sendmsg con gli opportuni parametri nell'ancillary data).

I test effettuati sono di due tipi:
- entrambe le app sulla stessa macchina utilizzando interfaccia di loopback (sbagliato);
- app eseguite su due macchine diverse sulla stessa rete locale.

Nel primo test effettuato sfruttavo un socket che aveva come indirizzo IPv6 l'interfaccia di loopback: così facendo i messaggi venivano ricevuti da recv.py ma ctypestest.py non riceveva alcun tipo di notifica (ovviamente).
Questo approccio non va bene in quanto un messaggio con destinazione il loopback della macchina non viene mai consegnato a livello data link.

A questo punto ho provveduto a far girare recv.py su un computer collocato sulla stessa rete locale della macchina con il custom kernel installato dove invece veniva eseguito l'app ctypestest.py.
I risultati di questo test sono piuttosto strani: in particolare la sendmsg ritorna l'errore di destination host unreacheable.
Ho così provato a diminuire il numero di pacchetti inviati (l'app ctypestest.py invia in loop numerosi pacchetti) fino ad inviare un solo messaggio ma l'errore persiste.
Inoltre provando a effettuare un ping verso qualsiasi indirizzo (interno alla rete locale e non) viene ritornato un messaggio ping: sendmsg: No buffer space available cosa che invece non avviene ripristinando i file originali del kernel 3.6.

Un eventuale problema potrebbe consistere nel fatto che l'inserimento del sequence number del frame 802.11 nella lista mantenuta con i relativi pacchetti tramite funzione ABPS_extract_packet_info non avviene nell'handler ieee802_tx_h_sequence ma bensì in un altro handler che dovrebbe occuparsi del power saving.