Progetto per il corso di Sistemi Mobili

Meccanismi di ritrasmissione anticipata su comunicazione wireless fra nodo mobile e access point

A.A. 2012/13

Marco Di Nicola

15 ottobre 2013

Indice

1 Introduzione **3**

2 Lavori precedenti **4**
2.1 TCP a ritrasmissione anticipata 4
2.2 Buffering e ritrasmissione di pacchetti su access point 5

3 Analisi e modifiche **7**
3.1 TCP a ritrasmissione anticipata 7
3.2 Buffering e ritrasmissione di pacchetti su access point 7

4 Dettagli tecnici **9**
4.1 TCP a ritrasmissione anticipata 9
4.2 Buffering e ritrasmissione di pacchetti su access point 11

5 Testing **14**

6 Conclusioni e sviluppi futuri **17**

1 Introduzione

Questo progetto ha come scopo l'analisi, il testing ed eventualmente la modifica di due sistemi preesistenti, atti a rendere più efficiente uno scambio di pacchetti a livello trasporto fra due end-systems, operando sul primo e ultimo hop della comunicazione.
Il primo di questi consiste nell'aggiunta di un meccanismo di ritrasmissione anticipata al protocollo TCP, che operi il rinvio del messaggio qualora una mancata notifica di avvenuta ricezione del frame data-link dal destinario venga segnalata, utilizzando un TED (Transmission Error Detector [1]) opportunamente modificato per gestire pacchetti TCP.
Questo sistema è implementato a livello di kernel del sistema operativo e permette di ridurre notevolmente i tempi agendo sul primo hop della comunicazione, quindi fra l'end-system mittente e l'access point al quale invia frames 802.11, rilevando più velocemente la necessità di ritrasmettere un segmento TCP e notificandola al soprastante livello di trasporto.

Il secondo sistema invece agisce sull'ultimo hop della comunicazione: il trasferimento dei frames data-link che incapsulano i pacchetti, dall'access point all'end-system destinatario.
Nel caso la trasmissione di un frame non vada a buon fine, quindi anche in questo caso in seguito alle segnalazioni di un TED in esecuzione, l'access point rispedisce i pacchetti IP all'host.
I pacchetti in uscita dal sistema operativo dell'access point vengono intercettati da Netfilter ed inseriti in una delle sue code. Questa coda verrà poi letta da un'applicazione dedicata, che li bufferizzerà in opportune strutture dati prima di trasmetterli all'host destinatario.

In uno scenario in cui le "ultime miglia" della comunicazione, intese come i segmenti di rete tra i due end-systems e i rispettivi access points, sono costituiti da canali wireless, l'utilizzo di questi due sistemi permetterebbe di ridurre notevolmente i tempi di attesa.
Sarebbe infatti possibile, nel caso di errori nella trasmissione in uno dei due sensi, prevaricare i più lunghi timers utilizzati dal livello di trasporto.

2 Lavori precedenti

Procediamo adesso con una panoramica sui risultati e/o limiti dei lavori precedenti alla soluzione di questi problemi.

2.1 TCP a ritrasmissione anticipata

Come base per il meccanismo di ritrasmissione anticipata nel protocollo TCP è stato utilizzato il progetto di tesi di Mirko Pedrini [2].
Il sopracitato lavoro fornisce un'implementazione della ritrasmissione in una specifica versione del kernel Linux (2.6.30-5), appoggiandosi, come precedentemente menzionato, ad un TED modificato per gestire anche i pacchetti TCP.
L'approccio utilizzato da Mirko consiste nell'assegnare un particolare identificativo alla principale struttura dati interna per la gestione dei pacchetti di rete nel kernel linux: sk_buff. Attraverso questo identificativo (skbID) è possibile effettuare un'associazione tra i frames a livello data-link e i segmenti TCP incapsulati al loro interno: esso viene impostato nella fase di creazione del segmento TCP e propagato attraverso lo stack di rete, eventualmente duplicandolo in caso di frammentazione del segmento in più pacchetti IP.

All'interno della funzione ABPS_info_response(), nel file *ABPS_mac80211.c* viene valutato lo stato attuale dell'invio di un frame (acked, ack non richiesto, ack non arrivato, ...) e, qualora sia rilevata una perdita, le informazioni necessarie alla ritrasmissione del pacchetto TCP (in particolare l'skbID) vengono inserite in una struttura ToT_Retransmit_Packet, la quale viene inserita in coda ad una lista di pacchetti da ritrasmettere. Viene quindi sollevato un evento di tipo ICSK_PREEMP_RETRANS, che indica una ritrasmissione anticipata.
Da notare che il meccanismo entra in funzione solo se il socket usato per trasferire i pacchetti è abilitato a leggere dalla sua coda degli errori, seppur l'approccio non consista nella notifica di perdita al livello applicativo (che quindi diventa il solo responsabile del rinvio) usata per i datagrams UDP.

L'evento sollevato al livello data-link viene infine gestito all'interno della funzione tcp_modified_timer(), all'interno del file *tcp_timer.c*.
Questa funzione prende come parametro il puntatore ad una struttura sock: la rappresentazione di un socket a livello network.

Viene prelevato l'elemento in testa alla lista di pacchetti da ritrasmettere ed il suo skbID è quindi confrontato con quello dei sk_buff contenuti nel campo sk_write_queue del socket, ossia la lista di pacchetti da trasmettere per quel socket.
Una volta individuato il segmento TCP il cui id corrisponde a quello da rinviare, questo viene ritrasmesso immediatamente, mediante chiamata alla funzione tcp_retransmit_skb().
Infine, la struttura ToT_Retransmit_Packet in testa alla lista viene eliminata, così che una prossima invocazione della funzione proceda a ritrasmettere il successivo segmento marcato come perso.

2.2 Buffering e ritrasmissione di pacchetti su access point

Per il sistema che opera su access point sono stati analizzati due lavori distinti: quello di tesi di Claudia Minardi [3] ed il progetto di Stefano Bettinelli e Vincenzo Gambale.
Entrambi utilizzano le API di Netfilter ed il tool **iptables** per catturare i pacchetti in transito ed effettuarne il buffering a livello applicativo, per poi gestirli in maniere differenti.

L'applicazione di Bettinelli e Gambale prevede la cattura dei pacchetti nella chain **INPUT** di iptables, quindi in fase di inoltro ai sockets locali nel sistema.
Dopo averli estratti dalla coda di Netfilter, questi vengono memorizzati in un buffer implementato come una lista e spediti in gruppi di 5 quando il contatore di elementi nel buffer raggiunge tale valore.
La spedizione avviene chiamando la funzione nfq_set_verdict() con il parametro indicante il verdetto pari a NF_ACCEPT, per ognuno dei 5 pacchetti, così da indicare a Netfilter di lasciarli transitare.
Poiché questa applicazione non consiste nell'effettiva ritrasmissione dei pacchetti, ma nel semplice buffering di questi ultimi, senza trattare situazioni di perdita, non è stata utilizzata come base per il lavoro attualmente in discussione.

Venendo invece al lavoro di Claudia, questo consiste nella cattura dei pacchetti nella chain **POSTROUTING** di iptables, quindi in uscita dal sistema.
Dopo essere stati copiati in una coda, il loro identificativo usato da Netfilter è passato alla funzione fake_sender(), la quale chiama nfq_set_verdict() con

verdetto NF_ACCEPT o NF_DROP (quest'ultimo ogni 9 pacchetti).
Con regolarità quindi si simula la notifica dal TED di un pacchetto non trasmesso con successo, provocando artificialmente la perdita dello stesso.
Se il "falso TED" (quindi la funzione appena menzionata) notifica la perdita di un pacchetto, questo viene estratto dal buffer e inviato attraverso un socket di tipo SOCK_RAW al destinatario specificato nell'header IP, utilizzando un thread separato.
E' necessario rinviarlo attraverso un nuovo socket poiché una volta stabilito un verdetto per il pacchetto Netfilter lo rimuove completamente dalla sua coda, restituendo errore nel caso di successivi riferimenti al suo identificativo.
Il pacchetto sarà quindi ricatturato in fase di uscita (chain POSTROUTING).
Per evitare pacchetti duplicati all'interno del buffer, in fase di aggiunta la coda viene attraversata, controllando che non ve ne sia uno con il medesimo identificativo.
Come chiave di identificazione univoca è utilizzata la coppia $< id, checksum >$: il checksum livello IP del pacchetto ed il campo id, utilizzato per distinguere diversi frammenti di un pacchetto più grande.
Infine, periodicamente il buffer (la cui dimensione non prevede limiti massimi) viene svuotato dei pacchetti meno recenti, così da ridurre il consumo di memoria.

3 Analisi e modifiche

L'obiettivo principale del progetto qui discusso è stato quello di analizzare le implementazioni descritte in precedenza, effettuare dei test e valutazioni dei risultati ed eventualmente modificarne alcuni aspetti ritenuti significativi.
Questa sezione descriverà in breve le valutazioni dei lavori precedenti e le eventuali modifiche apportate, lasciando alla prossima il compito di approfondire i dettagli implementativi.

3.1 TCP a ritrasmissione anticipata

L'analisi del codice che implementa la ritrasmissione anticipata del TCP è consistita fondamentalmente nel determinare quali fossero i segmenti TCP effettivamente ritrasmessi, in seguito ad una avvenuta perdita.
Per fare ciò è stata innanzitutto determinata la sequenza di chiamate di funzioni per gestione della rete, dislocate fra diversi sorgenti, poi il kernel linux è stato ricompilato con l'aggiunta di alcune `printk()` atte a determinare a run-time come fossero trattate le strutture dati implicate.

In sostanza, poiché il funzionamento effettivo è quello aspettato, non sono state apportate sostanziali modifiche al codice di Mirko (seguiranno descrizioni di alcuni test di correttezza dell'implementazione).

3.2 Buffering e ritrasmissione di pacchetti su access point

Veniamo adesso al buffering e ritrasmissione dei pacchetti su access point.
I limiti principali del lavoro di Claudia, come evidenziato nella sua documentazione, sono sintetizzati da questa serie di punti:

1. Fornire una più efficiente versione del buffer di pacchetti, consentendo una ricerca di duplicati più veloce.
2. Utilizzare un metodo alternativo per eliminare i pacchetti più vecchi dal buffer.
3. Utilizzare un vero meccanismo di notifica delle trasmissioni fallite, in sintesi: un vero TED.

Per quanto riguarda il punto **1**, si è fornita un'implementazione del buffer di pacchetti strutturata come una coda di dimensione limitata ed una hash-map per accesso agli elementi (puntatori a strutture che descrivono i pacchetti) in tempo costante.
Come chiave per accesso alla hash-map è stata utilizzata la stringa ottenuta concatenando il valore di **checksum** e **id** del pacchetto IP, in modo analogo a quanto fatto da Claudia.
Sussiste quindi la considerazione che, seppur non univoco, questo identificatore sia sufficiente in un sistema i cui dati utilizzati cambiano con rapidità.

Per il punto **2** invece è stata fatta una considerazione: il tempo massimo di residenza di un pacchetto nel buffer, secondo la visione di Claudia, è un'approssimazione in eccesso del massimo **TTL** (Time To Live) previsto per i pacchetti dal protocollo IP, quindi 64 hops, trasposto però in secondi.
Piuttosto che utilizzare un diverso intervallo temporale, si è optato per una soluzione alternativa: ricevere notifiche sui pacchetti effettivamente trasmessi (quindi il cui ACK sia stato ricevuto) dal TED, così da scartarli direttamente dal buffer.

Infine, per il punto **3**, è stata considerata l'idea di eseguire l'applicazione su un sistema con il kernel Linux 2.6.30-5 modificato con ritrasmissione anticipata, quindi abilitare anche per i pacchetti TCP la notifica tramite messaggio ICMP.
Oltre alla considerazione che tale meccanismo non sarebbe banale, data la gestione con garanzie di affidabilità del TCP che non rende i segmenti indipendenti l'uno dagli altri, occorrerebbe emettere verdetto DROP su tutti i pacchetti catturati da Netfilter, quindi rispedirli utilizzando un socket abilitato alla ricezione di messaggi di errore.
Inoltre, poiché ai fini della valutazione non è strettamente necessario utilizzare un vero TED, si è optato per una soluzione alternativa: rendere il numero di pacchetti persi variabile, utilizzando un generatore di numeri casuali piuttosto che un contatore che determinasse quali vadano scartati.

4 Dettagli tecnici

Verranno descritti adesso i principali aspetti implementativi delle modifiche apportate ai lavori precedentemente descritti e i più rilevanti dettagli tecnici.

4.1 TCP a ritrasmissione anticipata

Per prima cosa è occorso impostare un ambiente di testing adeguato, così da poter verificare l'effettivo funzionamento del meccanismo di ritrasmissione anticipata.
A questo scopo è stato generato, compilando i sorgenti della versione 2.6.30-5 del kernel Linux modificato, un pacchetto installabile utilizzando i seguenti comandi:

```
make menuconfig # menu grafico per selezionare i moduli da
    inserire nel nuovo kernel

make-kpkg clean # pulizia di files oggetto generati da precedenti
    compilazioni

fakeroot make-kpkg --initrd --append-to-version=-custom
    kernel_image kernel_headers # compilazione e creazione di
    package .deb di headers e immagine del kernel
```

Per la compilazione è stata utilizzata la versione 4.4 di *gcc*, poiché le successive restituivano errori di compilazione.
I package derivanti sono stati installati su una distribuzione Ubuntu 8.04 in esecuzione su macchina virtuale, creata utilizzando il tool di emulazione **VMWare Workstation**.
Nessuna delle distribuzioni più recenti testate, comprese alcune versioni di Debian, hanno consentito il boot del sistema operativo con il kernel modificato, probabilmente per vari problemi di retro-compatibilità.
Dato che VMWare Workstation non implementa una virtualizzazione di scheda di rete wireless, è stato necessario utilizzare un adattatore wifi usb per permettere al sistema di utilizzare il protocollo 802.11 e quindi accedere alle funzionalità interessate.

Venendo al codice, sono state aggiunte ulteriori stampe, mediante chiamate a printk(), in punti significativi dell'esecuzione: all'interno della funzione ABPS_info_response() (*ABPS_mac80211.c*), per determinare quali skb_buff fossero marcati per la ritrasmissione e quali no (usando come riferimento il loro skbID); in tcp_modified_timer() (*tcp_timer.c*), per avere un'indicazione di quali skb_buff fossero effettivamente ritrasmessi; in tcp_transmit_skb (*tcp_output.c*), per avere una visione globale di tutti i segmenti TCP passati al livello network, considerandone anche il sequence number.

Quindi l'output del comando dmesg -c su shell bash è stato salvato in un file per analizzare l'esecuzione, determinando che la ritrasmissione opera solamente sui singoli segmenti persi, senza ritrasmettere pacchetti superflui.
É stato utilizzato questo sistema, piuttosto che un semplice dump dei pacchetti in uscita con un analizzatore di traffico, in quanto offre risultati più chiari (non essendoci riferimento all'skbID nell'header dei pacchetti).
Sull'host destinatario invece è stato eseguito *tcpdump* per produrre un file con i pacchetti TCP arrivati dall'host mittente.
Questo file è poi stato analizzato con *wireshark*, così da effettuare un riscontro sui sequence numbers dei pacchetti stampati dalle printk() sull'altro host.

Per il testing sono state scritte due semplice applicazioni (nei files *retransmit_test_cli.c* e *retransmit_test_srv.c*) che consentono di attivare l'opzione IP_RECVERR di un socket TCP (così da godere del meccanismo di ritrasmissione anticipata) e trasmettere/ricevere pacchetti in numero e dimensioni variabili, specificate come input dall'utente.
Le due applicazioni sono eseguite su due end-systems diversi: una di esse si pone in listen() e l'altra effettua una connect(), quindi si scambiano un numero finito di messaggi.
Mediante chiamate a gettimeofday() si misura la durata della comunicazione (parametro poi utilizzato per valutare la bontà del sistema in fase di testing) con precisione di microsecondi.

Esempio di esecuzioni:

```
# su host A
./retransmit_test_srv 55556 2048 # parametri: [ porta d'ascolto ]
    [ dimensione payload messaggi ]

# su host B
./retransmit_test_cli 192.168.1.132 55556 2048 100
# parametri: [ indirizzo dst ] [ porta dst ] [ dimensione payload
    messaggi ] [ numero di messaggi ]
```

4.2 Buffering e ritrasmissione di pacchetti su access point

Fra le modifiche precedentemente menzionate, la più importante è sicuramente la gestione del buffer di pacchetti (files *packets_buffer.c* e *packets_buffer.h*).

La seguente struttura descrive il buffer di pacchetti:

```
typedef struct {
   unsigned int num_elems;
   pthread_mutex_t buffer_mutex;
   Packet *queue_head, *queue_tail;
   Packet **packets_hashmap;
} PacketsBuffer;
```

Le prime due variabili rappresentano rispettivamente un contatore del numero di elementi nel buffer ed una mutex per garantire Thread Safety nell'accesso alla struttura dati.
Seguono poi due puntatori al primo e ultimo pacchetto contenuto nella lista, in modo da consentire una veloce aggiunta in coda e rimozione da testa.
Il campo packets_hashmap, invece, costituisce l'hash-map utilizzata per verificare la presenza di un determinato pacchetto nella coda, a prescindere dalla sua posizione.

Non viene descritta la struttura Packet, essendo molto simile a quella definita nel progetto di Claudia.

Coda e hash-map hanno dimensioni differenti, specificate dalle costanti QUEUE_SIZE (4096) e MAP_SIZE (103997).
La funzione di hashing utilizzata è la seguente:

```
static unsigned long hash(PacketID* pkt_id) { ... }
```

A partire dall'identificativo di un pacchetto (descritto dalla struttura PacketID), la funzione calcola un valore di tipo long, utilizzando un algoritmo proposto da Dan Bernstein sulla community comp.lang.c.
Questo valore viene quindi modulato su MAP_SIZE (scelto come un grande numero primo in modo da ridurre la regolarità del modulo) ed il puntatore alla struttura Packet inserito nella corrispondente posizione in packets_hashmap.
Sono stati condotti alcuni test sul numero di collisioni nell'associazione di diversi identificativi dei pacchetti ai medesimi long, osservando che il loro numero non è rilevante, considerando la frequenza con cui dovrebbero cambiare i dati nel buffer (fino a 109 collisioni su 103997 combinazioni diverse di valori $< id, checksum >$, variando soprattutto il valore del checksum).
Operazioni definite sono la creazione e distruzione di un buffer, aggiunta, rimozione di un pacchetto (quest'ultima anche in posizione arbitraria) e recupero del suo puntatore.
Tutte le operazioni sono eseguite in tempo $O(1)$, minimizzando l'overhead computazionale che affliggeva il progetto di Claudia, in caso di traffico ingente.

Il finto TED è implementato come una semplice funzione che determina casualmente l'esito della "trasmissione" del pacchetto, scrivendo su una **pipe** letta dal thread retransmitter l'id del pacchetto da rinviare.
In base al valore restituito, la funzione chiamante (la stessa che funge da callback per l'estrazione di pacchetti dalla coda di Netfilter), decide se eliminare o meno il pacchetto dal buffer, chiamando remove_packet_byID().

```
static int fake_TED(struct nfq_q_handle *qh, uint32_t
    netfilter_packet_id, Packet* pkt) {
   if (packet_lost()) {
      nfq_set_verdict(qh, netfilter_packet_id, NF_DROP, 0, NULL);
      write(comm_pipe[1], &pkt->identifier, sizeof(PacketID));
      return ERROR_RETVALUE;
   }
   else {
      nfq_set_verdict(qh, netfilter_packet_id, NF_ACCEPT, pkt->len,
          (unsigned char*) pkt->content);
      return 0;
   }
}
```

La funzione packet_lost() genera un numero casuale nell'intervallo $[0, 1]$, utilizzato per determinare se il pacchetto rientra nella statistica di quelli persi o meno, con una frequenza che può essere assegnata come valore di input dall'utente, così da testare diverse tipologie di scenari.

```
static int packet_lost() {
   return ((float) random() / (float) RAND_MAX) <
         (packets_loss_rate / 100.0F);
}
```

Per avviare l'applicazione occorre eseguire come root i seguenti comandi, dando come parametri di *ap_retransmitter* la coda di Netfilter dalla quale estrarre i pacchetti (default: 10) e la frequenza relativa con cui perdere pacchetti (default: 10 %).

```
iptables -t mangle -A POSTROUTING -j NFQUEUE -o br0 --queue-num 10
./ap_retransmitter 10 20
```

5 Testing

Per concludere questo documento sono riportati e descritti alcuni risultati ottenuti in fase di testing dei due sistemi.
Come prima cosa occorre descrivere lo scenario utilizzato per i test:

Host A: Netbook (rappresentante il nodo mobile) con distribuzione Ubuntu 8.04 in esecuzione su macchina virtuale (VMWare), utilizzando il kernel linux 2.6.30-5 modificato per ritrasmissione anticipata del protocollo TCP. Come interfaccia di rete wireless è stato utilizzato un adattatore wifi usb ZyDAS. Esegue l'applicazione *retran_cli* (con ritrasmissione) o *cli* (senza ritrasmissione).

Access Point: Notebook con sistema operativo Debian 7 che funge da access point (con le regole di iptables precedentemente menzionate attive e l'applicazione *ap_retransmitter* in esecuzione per buffering ed eventuale ritrasmissione dei pacchetti). Per scopo di testing si è creato un secondo eseguibile, *ap_retransmitter_noretrans*, che si limita a perdere i pacchetti in uscita senza però ritrasmetterli. Per simulare le funzionalità di access point si è fatto uso dell'utility *hostapd*, in maniera del tutto simile a quanto fatto da Claudia nel suo lavoro e collegando interfacce wireless ed ethernet del notebook.

Host B: Sul notebook che fa da access point è anche attiva una seconda macchina virtuale che utilizza un secondo adattatore wifi usb ZyDAS per connettersi però ad un'altra rete. Questa virtual machine esegue l'applicazione *srv*, con ip pubblico e porta 55556 aperta sul router.

I tre attori appena descritti costituiscono lo scenario in cui l'host A (vm su netbook), associato via 802.11g ad un access point (notebook), instaura una connessione TCP con l'host B (vm su notebook, non connessa alla medesima LAN) e scambia un determinato numero di messaggi.
Per simulare una certa distanza fra i due host (difatti i gateway delle due reti -reale e notebook- utilizzate sono posti a distanza di un metro circa) è stata utilizzata l'utility *tc*, chiamando il seguente comando come root sui terminali di host A e B:

```
tc qdisc add dev wlan0 root netem delay 64ms # latenza aggiunta di
   64 ms
```

Il netbook che costituisce l'host A è stato quindi posto a buona distanza dal notebook usato come access point e spostato durante la trasmissione dei messaggi, così da aumentare il tasso di errori nella comunicazione.

Sono riportati quindi i risultati che mostrano il tempo medio, su 10 prove per caso, impiegato a spedire 200 pacchetti TCP di dati tra A e B (100 da A, ai quali B risponde), con diverse dimensioni del payload, utilizzando diverse combinazioni (n no / y si) dei sistemi precedentemente illustrati.

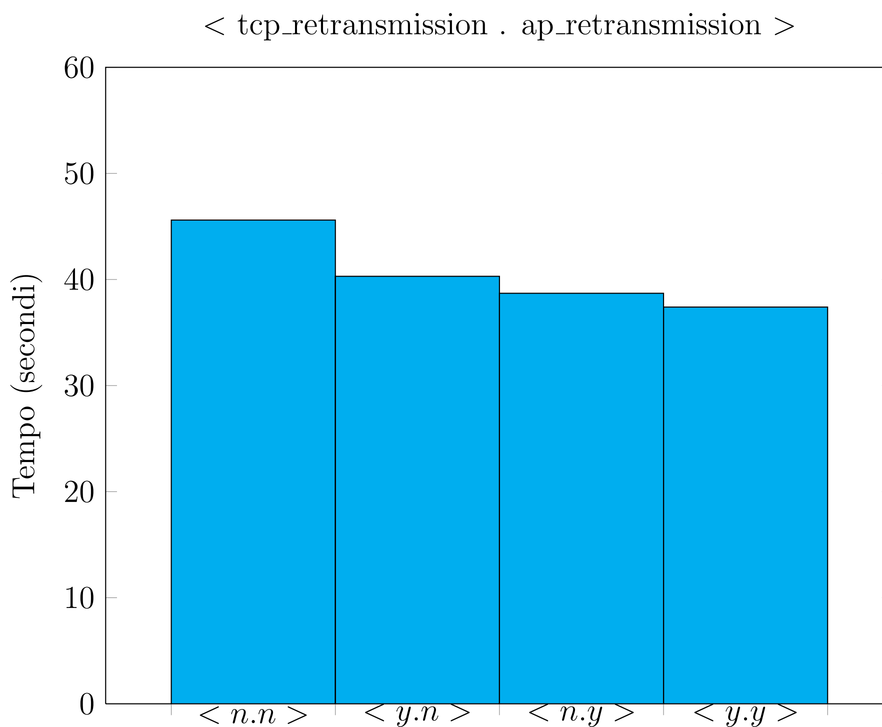

Figura 1: Dimensione del payload: 512 Bytes

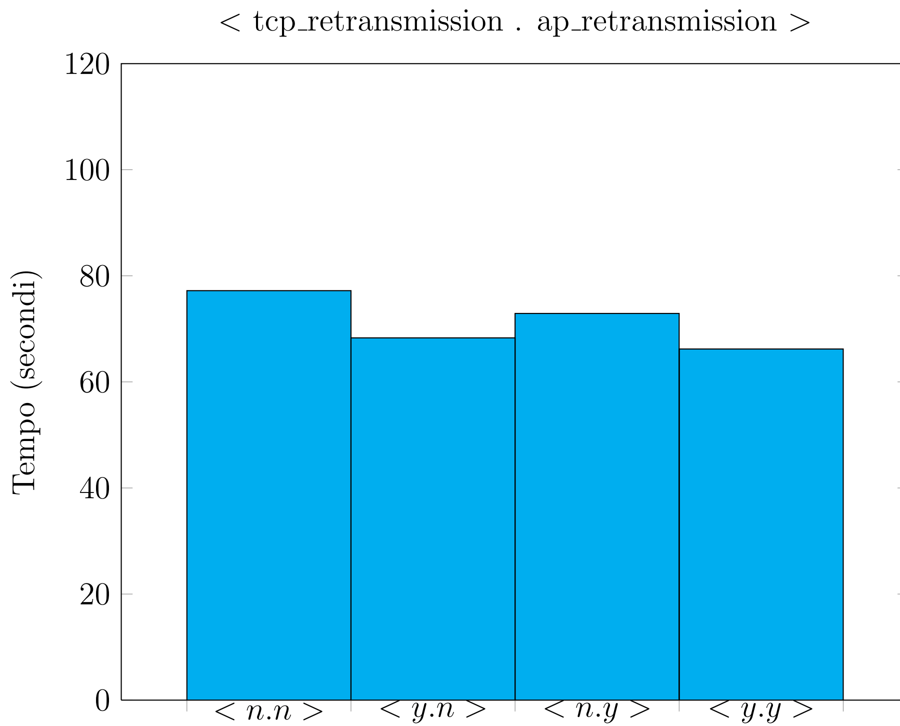

Figura 2: Dimensione del payload: 2048 Bytes

Possiamo osservare che con un payload di 512 Bytes è la combinazione $< n.y >$ (quindi buffering e ritrasmissione su AP), fra le due che prevedono l'utilizzo di uno solo dei sistemi, a dare risultati migliori.
Nelle prove con payload di 2048 Bytes è il contrario, probabilmente a causa di frammentazione (è in uso una MTU sul canale wireless pari a 1500 Bytes) e quindi maggiore invio/perdita di frames data-link.
Si ricorda infatti che la perdita di pacchetti trasferiti dall'access point al nodo mobile è provocata artificialmente, senza tener conto in alcun modo dello stato del canale.
In entrambi i casi è osservabile una effettiva diminuzione dei tempi qualora entrambi i meccanismi siano in funzione, rispettivamente su host A ed access point.

6 Conclusioni e sviluppi futuri

Per concludere possiamo menzionare alcuni interessanti sviluppi proposti per questo lavoro:

- Effettuare il porting della ritrasmissione anticipata sulla versione più recente del kernel Linux. Alcune problematiche iniziali sono sorte proprio a causa dell'utilizzo di una versione così vecchia, che difficilmente trova retro-compatibilità nei sistemi operativi attuali. Non si è stimato quanto possa essere complicato questo porting, ma è da notare come alcune strutture chiave implicate siano cambiate radicalmente (ad esempio sk_buff).

- Utilizzare un vero TED in congiunzione con il meccanismo di buffering e ritrasmissione su access point. Si può pensare di notificare, dal sistema operativo, anche le avvenute perdite di segmenti TCP; come menzionato in precedenza la cosa è tutt'altro che banale, soprattutto a causa della gestione interna del tcp e frammentazione di payload che provocano il superamento della MSS prefissata.

- Svolgere un numero maggiore di tests, allestendo scenari più significativi, con maggiori interferenze radio e che includano distanze da alta latenza.

Riferimenti bibliografici

[1] Vittorio Ghini, Giorgia Lodi, Fabio Panzieri.
Always Best Packet Switching: the mobile VoIP case study.
Dept. of Computer Science, University of Bologna, Italy.

[2] Mirko Pedrini.
TCP a ritrasmissione asimmetrica anticipata su WiFi.
Tesi di laurea discussa alla Facoltà di Scienze, Università Alma Mater Studiorum.
A.A 2011/2012

[3] Claudia Minardi.
TCP a ritrasmissione anticipata tramite Access Point Wi-Fi.
Tesi di laurea discussa alla Facoltà di Scienze, Università Alma Mater Studiorum.
A.A 2012/2013