UNIVERSITÀ DEGLI STUDI DI BRESCIA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

DIPARTIMENTO DI ELETTRONICA PER L'AUTOMAZIONE

Progetto, implementazione e caratterizzazione sperimentale di un algoritmo per il controllo della banda dei flussi TCP

Studente:

Marco Ghidinelli Matricola: **22757**

Relatore:

Prof. Luca Salgarelli

ANNO ACCADEMICO 2004/2005

«Non c'è vento favorevole

per il marinaio che non sa dove andare»

Lucio Anneo Seneca

Ringraziamenti

a luca.

che è un grande.

e mi ha proposto questa tesi bellissima.

senza tracciarmi mai la strada, ma facendomi capire dove cercarla.

«Se non c’è vento, rema.»

Proverbio polacco

Indice

Capitolo 1

Introduzione

La diffusione sempre più capillare della rete Internet ed i costi di connessione in costante diminuzione la stanno trasformando nel media universale di comunicazione del prossimo futuro.

Il continuo aumento della banda disponibile agli utenti finali non ha però distolto l'attenzione dalle politiche di gestione dell'accesso alla connessione Internet: i nuovi servizi che Internet fornisce oggi - voice over IP, video on demand o peer to peer, solo per citarne alcuni - sono fortemente diversificati e hanno differenti richieste di interattività, banda e ritardi massimi di trasmissione. È quindi prevedibile che, anche nel prossimo futuro, la gestione delle priorità di utilizzo della rete Internet rivestirà notevole importanza nella vita di tutti i giorni.

La quasi totalità del traffico Internet utilizza oggi il protocollo TCP, sviluppato all'inizio degli anni 80, che si basa sul meccanismo del controllo delle congestioni per determinare la banda che un trasmettitore può utilizzare. Tale meccanismo cerca semplicemente di utilizzare al massimo la capacità del canale a sua disposizione e ha come conseguenza quella di

non consentire una distribuzione controllata della banda che viene invece suddivisa a seconda di vari parametri fisici di funzionamento dei singoli flussi.

In particolare, come riportato da [1], si ottiene una proporzionalità inversa della banda allocata rispetto al Round Trip Time (RTT) del flusso in questione, dove il RTT rappresenta il tempo che intercorre dall'invio di un pacchetto alla ricezione della conferma della sua corretta trasmissione. Il risultato è quello di ottenere una distribuzione non ottimale della banda disponibile che risulta dipendente dai parametri fisici delle diverse connessioni.

Nel caso la connessione di rete sia fornita da parte di un ente o di una azienda questa impossibilità a controllare la banda disponibile diventa ancora più problematica: in situazioni limite alcuni utenti potrebbero utilizzare gran parte della banda di collegamente disponibile, addirittura per scopi ludici e non strettamente collegati a quelli per cui tale connessione viene fornita, andando di fatto a limitare notevolmente la qualità dei servizi che possono essere offerti attraverso questa connessione.

Risulta quindi evidente la necessità di metodologie in grado di permettere l'imposizione di politiche di gestione e di priorità ad una connessione di rete condivisa.

La figura 1.1 rappresenta lo scenario di rete considerato. Il collegamento è composto dal flusso di andata (dal trasmettitore al ricevitore) in cui vengono trasportati i dati, e dal flusso degli ACK di ritorno (dal ricevitore al trasmettitore) che segnalano lo stato della connessione.

La letteratura propone delle metodologie per la gestione di una connessione condivisa che permettono di assegnare caratteristiche arbitrarie

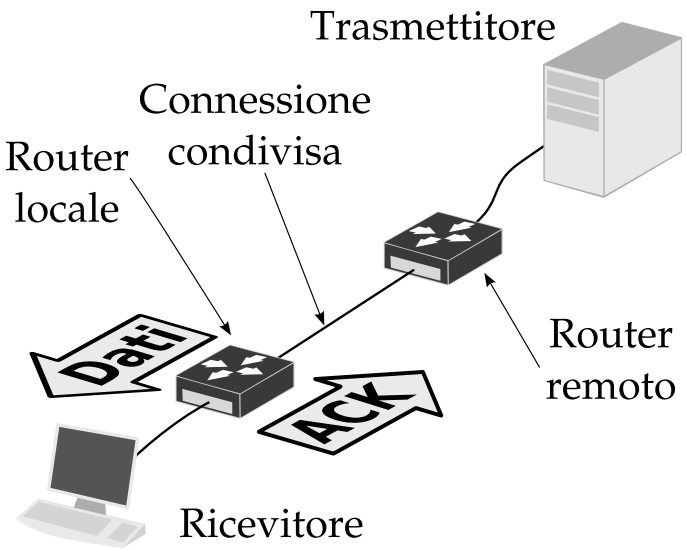

Figura 1.1: Configurazione di rete con banda limitata su un collo di bottiglia

ai flussi o alle classi di flussi che concorrono per l'assegnazione delle risorse del collegamneto. Queste prevedono che il controllo dei flussi TCP avvenga sul router che connette la rete locale con la connessione condivisa, discriminando in base al flusso di appartenenza dei pacchetti la loro priorità di utilizzo della connessione condivisa. Queste metodologie permettono inoltre la definizione di parametri formali di distribuzione della banda condivisa e attuano dei meccanismi atti a permettere l'interattività dei singoli flussi. Caratteristica comune di questi metodi è la loro implementazione sul flusso dati della connessione TCP, diretta dal trasmettitore al ricevitore nella figura 1.1. I pacchetti che arrivano al router locale sono allineati in varie code di tipo *First In First Out* (FIFO) sul router locale, da cui vengono poi rilasciati ordinatamente per essere trasmessi al ricevitore nella rete locale.

La limitazione di questi algoritmi è quella di dover memorizzare all'in-

terno del router le code dei pacchetti appartenenti al flusso dei dati inviati dal trasmettitore al ricevitore. Una volta esaurita la memoria del router infatti i nuovi pacchetti devono essere scartati e di conseguenza si necessiterà la ritrasmissione da parte del trasmettitore.

In [2] è stato proposto un algoritmo alternativo per il controllo del traffico del flusso dati di una connessione TCP che agisce sulla coda dei pacchetti ACK di riconoscimento che vengono mandati dal ricevitore al trasmettitore.

Le dimensioni dei pacchetti ACK sono mediamente del 95% più piccole rispetto a quelle dei pacchetti dati a cui si riferiscono, e permettono una notevole diminuzione della memoria utilizzata all'interno del router che si traduce di conseguenza nella possibilità di poter controllare un maggiore numero di flussi a parità di memoria disponibile nel router locale.

Lo scopo di questa tesi è stato quello di determinare i requisiti e le specifiche di funzionamento dell'algoritmo, di implementare all'interno del kernel di Linux l'algoritmo, e di determinare la validità dell'implementazione svolta mediante una analisi dei risultati sperimentali ottenuti.

L'analisi di tali risultati dimostrerà che l'implementazione creata permette effettivamente di controllare il traffico utilizzato dai flussi dati agendo solo sugli ACK del medesimo flusso.

L'algoritmo proposto permette inoltre di porre le basi per una futura implementazione di un controllo del traffico che miri a modificare la dimensione della finestra di ricezione dichiarata dal ricevitore, permettendo quindi un controllo migliore della banda utilizzata dal flusso dati.

1.1 Struttura della tesi

La tesi è suddivisa in sei capitoli. Dopo questa introduzione, nel secondo capitolo verrà fornita una panoramica sulla rete TCP/IP e sugli algoritmi per il controllo della congestione in esso implementati. Verranno poi presentati alcuni algoritmi standard per la gestione di un collegamento di rete condiviso.

Nel terzo capitolo viene fornita una breve panoramica su Linux e sull'implementazione del controllo del traffico da esso fornito. Verranno in seguito analizzate problematiche e soluzioni adottate relativamente alla programmazione all'interno del kernel di questo sistema operativo.

Nel quarto capitolo, dopo una breve illustrazione del principio di funzionamento dell'algoritmo Enhanced TCP ACK Pacing (ETAP), verrà analizzata l'implementazione eseguita di tale algoritmo all'interno del kernel Linux.

Nel quinto capitolo verranno presentati i risultati sperimentali ottenuti dall'algoritmo ETAP in alcuni test mirati ad evidenziarne il comportamento in tipiche modalità di funzionamento. Verranno quindi analizzate le prestazioni alla luce dei requisiti inizialmente formulati.

Nel sesto capitolo infine vengono riassunte le conclusioni del lavoro svolto e verranno analizzati i limiti di questa implementazione e i possibili sviluppi futuri.

Capitolo 2

Panoramica sul controllo del traffico nelle reti a pacchetto

2.1 Il Protocollo IP

L'Internet Protocol (IP) è un protocollo di comunicazione utilizzato per trasferire dati da una sorgente a una destinazione attraverso delle reti di interconnessione a commutazione di pacchetto. I dati da inviare vengono separati in blocchi denominati pacchetti IP e instradati dal mittente al destinatario, dove mittente e destinatario vengono identificati da un indirizzo IP univoco.

Le funzionalità del protocollo IP sono state volutamente limitate alla sola capacità di instradare pacchetti attraverso la rete ignorando completamente ogni meccanismo per aumentare l'affidabilità della trasmissione. Nessun controllo viene eseguito sulla qualità o sull'ordinamento dei pacchetti inviati, che di conseguenza possono andare perduti o venire duplicati o raggiungere il destinatario in un ordine diverso da quello in cui sono

stati inviati. L'affidabilità della trasmissione viene di solito demandata a dei protocolli di livello superiore che vengono trasportati all'interno dei pacchetti IP.

Il protocollo IP viene spesso definito come *connectionless, stateless, unreliable*:

Connectionless, senza connessione. Il termine connectionless descrive una comunicazione tra due punti nella quale un messaggio può essere inviato da un punto all'altro senza alcun accordo preliminare. I pacchetti IP vengono infatti trasmessi dal mittente senza che venga eseguito alcun controllo relativo alla disponibiltà del destinatario a riceverne. In caso di indisponibilità del ricevitore, il pacchetto sarà scartato da un router intermedio dovrà essere reinviato.

Stateless, senza stato. Il ricevitore di una trasmissione IP non ha modo di determinare in che fase della comunicazione si trovi e non ha alcun modo di conoscere l'ordinamento dei pacchetti che sta ricevendo, se non affidandosi ad informazioni aggiuntive esterne al protocollo stesso.

Unreliable, inaffidabile. La trasmissione dei pacchetti IP può andare o meno a buon fine ma non vi è modo ne per il trasmettitore ne per il ricevitore di determinare se tutti i pacchetti inviati sono arrivati a destinazione.

I pacchetti IP vengono instradati nel loro percorso dal sorgente al destinatario da router che rendono possibile la comunicazione tra reti distinte. Le limitate funzionalità del protocollo IP si traducono nella semplicità di

costruzione di tali router che possono quindi concentrare le loro risorse sulle prestazioni e sulla gestione delle interconnessioni delle reti.

La versione 4 del protocollo IP, definita in [3] e denominata IPv4, è stata la prima universalmente distribuita e su di essa è stata costruita la rete Internet. Tale versione consta di indirizzi numerici a lunghezza fissa di 32 bit permettendo quindi 4'294'967'296 indirizzi IP distinti.

Un generico pacchetto IP è composto da due parti: l'intestazione (header), che contiene le informazioni necessarie al suo corretto instradamento, e la parte dati (payload). Nell'header trovano posto gli indirizzi di mittente e destinatario del pacchetto, la sua dimensione comprensiva sia header che di parte dati, la versione di protocollo utilizzata e altre informazioni.

2.2 Il protocollo TCP e Il controllo delle congestioni

Il protocollo di controllo della trasmissione (TCP), definito in [4], è stato ideato e sviluppato contestualmente al protocollo IP, in modo da integrare le caratteristiche che erano non sono state volutamente inserite in quest'ultimo. Mediante il TCP, le applicazioni di rete possono creare delle connessioni sulle quali scambiarsi dati: il protocollo garantisce una consegna ordinata e sicura dei dati trasmessi dal mittente al ricevitore.

Il protocollo TCP è implementato al livello superiore del protocollo IP e si appoggia a quest'ultimo per la consegna effettiva delle informazioni.

Quando il TCP riceve da una applicazione un flusso di byte da inviare in rete, lo suddivide in pacchetti di dimensione appropriata e fornisce questi pacchetti al protocollo IP. Il TCP controlla che nessuno dei pacchetti vada perso fornendo ad ognuno un numero di sequenza *sequence number*

che è anche utilizzato per verificare l'ordine con cui i pacchetti giungono a destinazione. Ad ogni pacchetto viene inoltre aggiunta una informazione di *checksum* che permette al ricevitore di verificare la correttezza dei dati ricevuti a fronte di errori di trasmissione.

Il modulo TCP del ricevitore rimanda indietro mediamente ogni 2 pacchetti ricevuti un pacchetto di riconoscimento, chiamato *acknowledgement* (ACK), contenente il sequence number della parte di trasmissione ricevuta correttamente. Il trasmettitore in questo modo viene notificato della eventuale perdita di alcuni pacchetti durante la trasmissione e può quindi dopo un determinato tempo provvedere alla ritrasmissione dei pacchetti perduti.

Il TCP permette inoltre di creare più connessioni distinte tra gli stessi end-point della comunicazione, aggiungendo l'informazione del numero di porta del ricevitore e del trasmettitore al pacchetto in modo da identificare univocamente un flusso TCP mediante le coppie indizzo/porta del ricevitore/trasmettitore.

Tali informazioni sono contenute all'interno dell'header TCP che precede la parte dati del pacchetto TCP stesso. Sia l'header che il segmento dati del pacchetto TCP sono trasmessi nel campo dati del pacchetto IP che li contiene.

I pacchetti di ACK ricevuti e il loro istante di arrivo sono l'unica informazione che arriva al trasmettitore relativamente allo stato della trasmissione in corso e gli permettono di inferire le condizioni della rete nel tratto di trasmissione. La prima implementazione del controllo delle congestioni implementata in[4] è stata migliorata successivamente con l'introduzione di nuovi algoritmi che permettessero di sfruttare meglio il canale di comunicazione.

2.2.1 Slow Start e Congestion Avoidance

Gli algoritmi di *slow start* e *congestion avoidance* sono stati introdotti da Van Jacobson [5] per controllare la quantità di nuovi dati da trasmettere in rete. L'implementazione di questi algoritmi necessita la definizione di alcuni parametri che devono essere mantenuti per ogni flusso TCP.

Sequence Number è il numero progressivo contenente il numero di byte trasmessi fino al primo byte del pacchetto corrente.

Acknowledgment Number contiene il valore dell'ultimo sequence number ricevuto dal ricevitore e ritornato al mittente incrementato di uno.

Sender Maximum Segment Size è la dimensione massima del pacchetto che può essere inviato dal trasmettitore. Il suo valore viene determinato nella fase di instaurazione del flusso TCP.

Receiver Window, *rwnd* rappresenta la quantità massima di dati che il ricevitore è in grado di processare ed è comunicata attraverso gli ACK di ritorno dal ricevitore.

Congestion Window, *cwnd* è una variabile che limita la quantità di dati che il trasmettitore può inviare. La *cwnd* viene modificata dal trasmettitore in base alla banda e alle condizioni di traffico, e non può mai superare la dimensione della *rwnd*. Il suo valore iniziale deve essere al più di due SMSS.

Flight Size indica le quantità di dati che il trasmettitore TCP ha inviato, ma di cui non ha ancora ricevuto conferma dagli ACK di ritorno.

Slow Start Threshold, ssthresh definisce se si dovrà utilizzare l'algoritmo

di Slow Start o di Congestion Avoidance per determinare la quantità di nuovi dati da inviare. Se la *cwnd* è inferiore a ssthresh viene utilizzato Slow Start, viceversa si utilizza Congestion Avoidance.

Round Trip Time, RTT rappresenta il tempo che intercorre dalla trasmissione di un pacchetto alla ricezione del pacchetto di *acknowlegdement* ad esso relativo.

Retransmission Time Out, RTO indica il tempo dopo il quale si può considerare perso un pacchetto inviato di cui non si è ricevuto il relativo pacchetto di *acknowledgement*. Definito da Paxton e Allman in [6].

Slow Start è utilizzato per determinare la capacità massima del canale quando si inizia una nuova connessione. Durante la fase di Slow Start, TCP incrementa *cwnd* al più di un SMSS ogni ACK ricevuto che riconosca nuovi dati. Questo comportamento fa crescere esponenzialmente la *cwnd* in modo da raggiungere rapidamente la capacità massima della connessione. La fase di Slow Start termina quando si ha una perdita di pacchetti evidenziata dal mancato ritorno degli ACK relativi a pacchetti già trasmessi.

In seguito al raggiungimento della massima capacità del canale il trasmettitore utilizza l'algoritmo di Congestion Avoidance. Questo algoritmo è molto meno aggressivo nell'incrementare la *cwnd* in quanto questa viene aumentata di un segmento ogni Round Trip Time.

Ogni volta che un ACK viene ricevuto, la Flight Size diminuisce del valore riconosciuto da questo ACK, permettendo quindi la trasmissione di uno o più segmenti di nuovi dati. In questo modo il tempo di trasmissione viene scandito dal ritmo dei pacchetti ricevuti permettendo di sfruttare ef-

ficacemente la capacità della rete senza eccedere in picchi di traffico e nelle conseguenti ritrasmissioni.

Nel caso di perdita di un pacchetto, evidenziata dalla non ricezione dell'ACK relativo entro il tempo di Retransmission Time Out, l'algoritmo prevede di portare il valore di ssthresh ad essere pari alla metà della Flight Size e il valore di *cwnd* pari a un SMSS. Viene quindi ritrasmesso il segmento perduto e si ricomincia con l'algoritmo di Slow Start.

2.2.2 Fast Retransmit e Fast Recovery

Questi algoritmi, definiti in [7], sono un'estensione dell'implementazione precedente, atte a evitare che in seguito alla rilevazione della perdita di un pacchetto, l'algoritmo debba ricominciare con la fase di Slow Start riducendo quindi la banda utilizzata.

Per un corretto funzionamento di Fast Retransmit e Fast Recovery tutte le implementazioni di TCP devono segnalare la ricezione di un segmento fuori ordine mediante l'invio immediato di un ACK duplicato, ovvero un segmento che vada a riconoscere lo stesso Sequence Number di un pacchetto precedentemente inviato. In questo modo il ricevitore segnala al trasmettitore l'eventuale perdita di un pacchetto.

L'algoritmo di Fast Retransmit interpreta l'arrivo di tre ACK duplicati come la perdita di un pacchetto e provvede quindi alla ritrasmissione dello stesso senza dover attendere la scadenza dell'RTO.

Dopo Fast Retransmit si passa alla fase di Fast recovery per la trasmissione di nuovi dati: la ricezione degli ACK duplicati indica infatti che il ricevitore sta ricevendo altri segmenti, che hanno quindi lasciato la rete.

Si pone quindi il valore di ssthresh pari alla metà della Flight Size e si ritrasmette il segmento perso.

A questo punto il ricevitore sicuramente ha ricevuto almeno tre pacchetti e quindi si pone la *cwnd* al valore della ssthresh e si somma ad essa un numero di segmenti pari al numero di ACK duplicati ricevuti (almeno tre, quindi). A questo punto se la finestra di congestione lo consente si trasmette un nuovo pacchetto.

Quando viene ricevuto un ACK che riconosce nuovi dati, si pone la *cwnd* al valore della ssthresh calcolato prima. Questo pacchetto dovrebbe riconoscere tutti i segmenti intermedi inviati tra il segmento perso e la ricezione del terzo ACK duplicato.

Nel grafico 2.1 è rappresentato un andamento esemplificativo del valore della finestra di congestione *cwnd* e della *ssthresh* di un trasmettitore TCP.

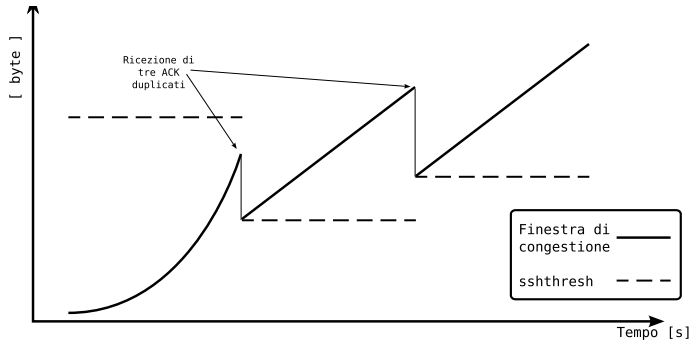

Figura 2.1: Andamento temporale della finestra di congestione *cwnd* e della *ssthresh* in una connessione TCP

Nel primo tratto la finestra di congestione cresce esponenzialmente utilizzando *slow start* fino a superare i limiti del canale di trasmissione e causando la perdita di alcuni pacchetti. Questa perdita viene segnalata al trasmettitore attraverso la ricezione di tre ACK duplicati inviati dal ricevitore.

In conseguenza di questo il trasmettitore riduce la sua *ssthresh* e si pone nella fase di *congestion avoidance*.

2.3 Gestione delle risorse nelle reti a pacchetto

Il protocollo TCP è stato ideato per cercare di utilizzare al meglio un singolo collegamento di rete cercando di massimizzare la sua banda di trasferimento. Quando però molti flussi devono condividere lo stesso canale trasmissivo, si ottiene analiticamente [1] che i parametri che concorrono alla determinazione della banda assegnata ad un flusso sono di tipo fisico. In particolare si ottiene una proporzionalità inversa della banda allocata rispetto al Round Trip Time (RTT) del flusso considerato, che implica quindi una distribuzione non ottimale della banda disponibile. Da questo la necessità di politiche di gestione della banda indipendenti dalle caratteristiche fisiche della connessione dei singoli flussi ma bensì basate su politiche di condivisione di link gestite localmente.

2.3.1 Fair Queueing

L'approccio proposto da Demers Kerhav e Shenker in [8] consiste nell'utilizzo di algoritmi di accodamento implementati nel gateway di connessione che permettano di gestire le situazioni di congestione dei flussi controllati.

Le funzioni di accodamento dei pacchetti determinano infatti il modo in cui avvengono le interazioni tra gli end-host dei flussi controllati che di conseguenza possono influenzare indirettamente anche il comportamento degli algoritmi di controllo della congestione dei trasmettitori. Questi

infatti possono essere indotti a inferire una congestione della rete a causa del rallentamento dei pacchetti inviati e di conseguenza a ridurre la loro finestra di congestione diminuendo quindi la banda utilizzata.

L'implementazione di tale approccio prevede la separazione in code FIFO dei pacchetti in base al flusso di appartenenza degli stessi: ogni nuovo pacchetto n ricevuto viene accodato alla relativa FIFO e viene memorizzato con esso il suo tempo di arrivo $T_{i,n}$.

L'intervallo di tempo necessario per inviare un pacchetto di dimensione P_n attraverso una connessione in grado di trasmettere a μ byte al secondo è determinato da $T_{p,n} = \frac{P_n}{\mu}$. Questo tempo viene sommato al tempo di arrivo del pacchetto $T_{i,n}$ e memorizzato in $T_{f,n} = T_{p,n} + T_{i,n}$. A questo punto l'algoritmo invia i pacchetti presi dalle varie code ordinandoli per il relativo $T_{f,n}$ associato.

Questo approccio permette di allocare in modo equo la banda condivisa tra flussi TCP e in particolare riesce a fornire una priorità maggiore ai pacchetti di piccole dimensioni, generalmente appartenenti a protocolli interattivi quali *telnet* o *ssh*, ma non permette alcun controllo arbitrario della banda utilizzata da un singolo flusso.

2.3.2 Class Based Queueing

L'approccio proposto in [9] da Floyd e Jacobson consiste nell'abilitare il gateway locale a controllare la distribuzione della banda della connessione di rete condivisa in base solo a parametri arbitrari forniti al gateway.

In particolare questo approccio prevede la creazione di una struttura gerarchica dove vengano definite delle classi di traffico arbitrarie, ad ognuna delle quali viene associate una coda di pacchetti.

Si prevede inoltre che per ogni classe siano definite le relative politiche di divisione della banda disponibile in condizioni di congestione.

Nella figura 2.2 viene mostrato un esempio di struttura di *link-sharing* dove i requisiti di banda allocati a ogni classe sono espressi in percentuale della banda totale disponibile.

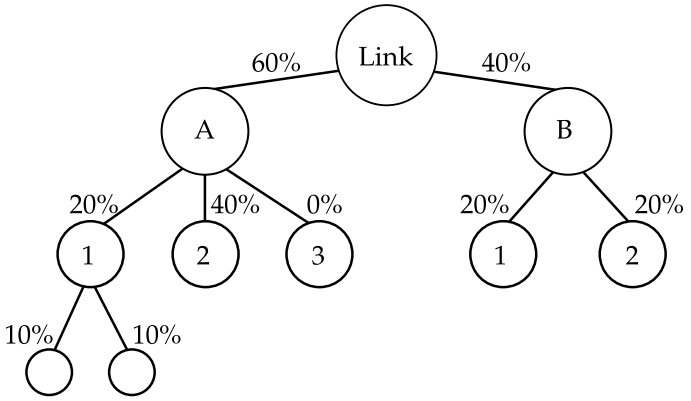

Figura 2.2: Esempio di una struttura di *link-sharing* gerarchica

Nessuna limitazione è fissata sulla tipologia semantica delle classi, che possono essere create in funzione di indirizzi IP, servizi o altro.

L'implementazione di questo sistema richiede alcune definizioni preliminari per la comprensione dell'algoritmo utilizzato. Alcune di queste sono diventate di uso generale nella nomenclatura del controllo del traffico.

General Scheduler, Link-Sharing Scheduler CBQ prevede che nel gateway siano disponibili due algoritmi di scheduling dei pacchetti che vengano applicati a seconda del caso alle code presenti. Il *general scheduler* funziona indipendentemente dallo stato delle varie classi presenti, mentre il *link-sharing scheduler* schedula i paccetti dalle classi che hanno ecceduto le limitazioni a loro imposte.

Regulated class, Unregulated class Una classe viene detta *regulated* se i pacchetti contenuti nella sua coda sono schedulati dal *link-sharing scheduler*, mentre viene definita *unregulated* se invece il suo traffico è associato al *general scheduler*.

Classifier Il *classifier* associa i nuovi pacchetti che arrivano al gateway alla coda della relativa classe di appartenenza.

Estimator L'*estimator* stima la banda utilizzata da ogni classe in un determinato intervallo di tempo permettendo di determinare il comportamendo della classe relativamente ai limiti di link-sharing definiti.

Overlimit, Underlimit, At-limit Una classe si definisce *overlimit* se ha utilizzato più banda di quanto le era stato allocato, *underlimit* se ha usato meno di una frazione specificata del suo limite, *at-limit* altrimenti.

Satisfied, Unsatisfied Una classe è definita come *unsatisfied* se si trova nello stato di *underlimit* da un determinato periodo di tempo. Viceversa è definita *satisfied*.

Levels Tutte le classi foglia di una struttura *link-sharing* appartengono per definizione al livello 1, e ogni classe interna ha il livello incrementato di uno rispetto a quello delle sue classi figlie.

L'algoritmo teorico di funzionamento di CBQ è il seguente:

Una classe può continuare a essere *unregulated* se è verificata almeno una delle due condizioni seguenti:

- La classe non è in *overlimit*, OPPURE

- La classe ha un ha un antenato non *overlimit* al livello *i*, e la struttura di *link-sharing* non ha classi *unsatisfied* in nessun livello minore di *i*.

Viceversa, la classe in questione sarà *regulated* dal *link-sharing scheduler*.

Una classe continuerà ad essere regolata finché è verificata almeno una delle due condizioni seguenti:

- La classe è *underlimit*, OPPURE
- La classe ha un antenato *underlimit* al livello *i*, e la struttura di *link-sharing* non ha classi *unsatisfied* a livelli minori di *i*.

La definizione dei parametri formali della condivisione utilizzato in CBQ ha posto le basi corrette per un approccio al controllo della distribuzione di banda in un link condiviso. La limitazione della sua implementazione è quellla di poter lavorare solo con la coda dei pacchetti dati, andando di conseguenza a utilizzare una notevole quantità di memoria per memorizzare le code dei pacchettti dei flussi da regolare.

L'algoritmo ETAP, la cui implementazione è oggetto di questa tesi, applica il controllo solo ai pacchetti ACK di ritorno riducendo di molto la memoria utilizzata dal router.

Capitolo 3

Networking avanzato con Linux

In questo capitolo verranno forniti i prerequisiti tecnici necessari alla comprensione dell'implementazione dell'algoritmo che verrà descritta nel capitolo successivo.

3.1 Risorse

Il kernel di Linux può essere scaricato liberamente dal sito `http://www.kernel.org/`, e una volta espanso il controllo del traffico risiede nella directory `net/sched/`.

Il programma utente *tc* può essere scaricato dal sito `http://developer.osdl.org/dev/iproute2/`. La versione di riferimento per questa tesi è la `2.6.11-050330` (Si noti che tale versione è per pura coincidenza simile a una versione del kernel).

3.2 Il sistema operativo Linux

Linux è un sistema operativo creato da Linus Torvalds nel 1991 e successivamente migliorato con la collaborazione di sviluppatori da tutto il mondo. Originariamente creato per l'architettura Intel 80386 è stato portato verso numerose altre piattaforme.

L'architettura di Linux è quella di un kernel monolitico, che fornisce cioè una astrazione completa dell'hardware attraverso un insieme di chiamate di sistema, le *system calls*, che consentono ai programmi utente di interfacciarsi con il livello hardware sottostante. A differenza dei kernel monolitici standard però molti driver di periferiche sono presenti sotto forma di moduli che possono essere caricati e scaricati mentre il kernel è in esecuzione. Questi moduli diventano parte integrante del kernel e ne condividono lo spazio di indirizzamento.

Una ulteriore differenza rispetto a un kernel monolitico è che in particolari situazioni anche il kernel stesso è interrompibile da altro codice kernel. Questa caratteristica implementata nella ultima release rende quindi una architettura monoprocessore molto simile dal punto di vista della programmazione ad una multiprocessore simmetrica.

Il kernel Linux è scritto nella sua quasi totalità in linguaggio C, associato a brevi segmenti di assembler, ed è rilasciato sotto licenza GNU General Public License.

Il kernel Linux implementa nativamente il protocollo TCP/IP sin dalle prime versioni: il suo stack TCP/IP è stato riscritto più volte fino ad ottenere una implementazione molto stabile e affidabile.

3.3 Il controllo del traffico in Linux

Il kernel Linux offre nativamente una serie di procedure e di servizi per consentire di implementare delle politiche di controllo del traffico sin dalla versione 2.2. Tale implementazione, ad opera di Alexey Kuznetsov, è interna al kernel stesso per ovvie ragioni di prestazioni, e viene gestita mediante dei comandi utente forniti al kernel attraverso il client a riga di comando *tc*.

La figura 3.1 rappresenta un router Linux su cui viene implementato il controllo del traffico.

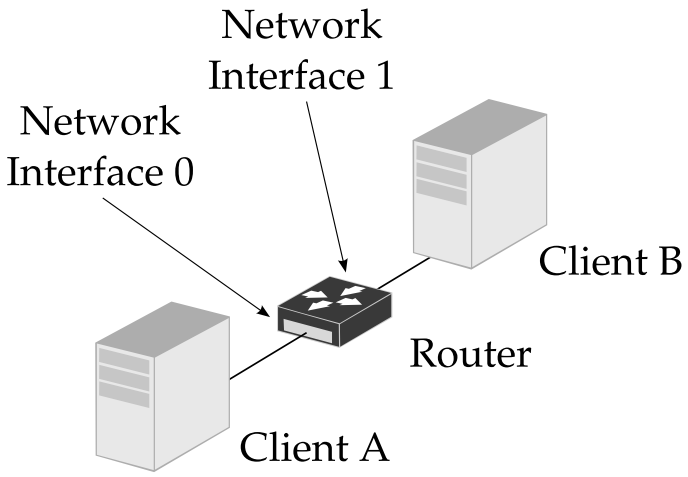

Figura 3.1: Controllo del traffico implementato mediante Router Linux

L'implementazione del controllo del traffico del kernel di Linux viene associata alle varie interfaccie di rete presenti, e può controllare solo i pacchetti nella coda di uscita di tale interfaccia. È quindi necessario determinare l'interfaccia corretta sulla quale configurare il sistema di controllo di rete. Considerando per esempio la figura 3.1, se si necessita di controllare il traffico proveniente dal Client B al Client A è necessario impostare tale

controllo sull'interfaccia *Network Interface 0* del Router.

Il controllo del traffico in Linux è composto dai seguenti componenti concettuali:

- discipline di accodamento;
- classi (all'interno di una disciplina di accodamento);
- filtri.

Ogni interfaccia di rete presente nel sistema è associata ad una *disciplina di accodamento* (qdisc) che controlla come la coda di pacchetti in uscita dall'interfaccia viene gestita. All'avvio il kernel inizializza tutte le interfaccie di rete con una delle qdisc più semplici, la `sch_fifo`, che semplicemente accoda tutti i pacchetti pronti ad essere trasmessi e li invia nell'ordine in cui sono arrivati. Attraverso il comando *tc* è possibile sostituire la disciplina di accodamento di default con un'altra.

Quando si rende disponibile un nuovo pacchetto da trasmettere attraverso una interfaccia di rete, viene richiamata la funzione di accodamento associata alla qdisc dell'interfaccia in questione, e le viene fornito il pacchetto.

L'implementazione di tale funzione è a completa discrezione della disciplina di accodamento scelta, che ha quindi la assoluta libertà di decidere la sorte del pacchetto.

Quando poi l'interfaccia di rete si trova nella possibilità di poter inviare un pacchetto verso l'esterno, viene richiesto alla qdisc associata ad essa di fornire un pacchetto. Anche in questo caso la qdisc ha la piena libertà di scegliere che pacchetto restituire, o eventualmente di non restituire alcun pacchetto.

Una disciplina di accodamento più elaborata può utilizzare delle *classi* di pacchetti diversi e processare in modo diverso ciascuna classe, per esempio assegnando una priorità maggiore ad una classe rispetto che ad un'altra. La creazione di una classe implica l'utilizzo di discipline di accodamento interne, le *inner qdisc*, in cui i pacchetti vengono accodati a seconda della classe di appartenenza. Non esiste alcuna limitazione sulla semantica delle varie classi interne.

Nelle discipline a classi, per determinare verso quale `qdisc` interna un pacchetto debba essere accodato, viene associata alla disciplina una lista di filtri che definiscono al loro interno dei parametri a cui i pacchetti vengono confrontati.

Se l'esito del confronto è positivo il pacchetto la scansione dei filtri termina e il pacchetto viene accodato alla classe definita dal filtro.

Esistono numerose tipologie di filtri che permettono di analizzare sia il contenuto dei vari header del pacchetto sia altre informazioni o metadati eventualmente aggiunti da altri sottosistemi del kernel.

Le discipline di accodamento a classi normalmente non conservano i pacchetti al loro interno ma li instradano verso delle discipline di accodamento interne, che a loro volta possono essere delle classi, che a loro volta possono inoltrare i pacchetti verso altre discipline di accodamento. Questa modularità strutturale unita alla versatilità dei filtri, che permettono di indirizzare verso qualunque classe o disciplina, rendono questa implementazione estremamente flessibile.

Solo le discipline a classi, denonimate anche *classful*, possono avere dei filtri al loro interno che indirizzino dei pacchetti verso particolari qdisc interne. Le discipline che non permettono la presenza di sottodiscipline in-

terne sono dette *classless*, e a tali discipline di accodamento non è possibile associare filtri.

Le discipline sono identificate attraverso un numero a 32 bit diviso in due parti da 16 bit cadauno, la parte *major* e la parte *minor* che deve essere univoco per ogni disciplina. In particolare, tutte le sottodiscipline di una *classful* qdisc devono condividere con la disciplina principale questo numero. Inoltre, il numero zero è riservato appunto alla *classful* qdisc radice.

Linux implementa nativamente praticamente tutte le discipline di accodamento standard e consente di aggiungerne di nuove in modo semplice e lineare, grazie alla sua architettura estremamente modulare.

Vediamone brevemente alcune, distinguendo quelle *classless* da quelle *classful*

3.3.1 Discipline di accodamento senza classi

Packet fifo e Bytes fifo, pfifo e bfifo. Sono delle discipline di accodamento implementate mediante una semplice lista first-in first-out e non accettano alcun parametro se non la lunghezza massima della coda misurata rispettivamente in pacchetti o in byte.

Random Early Detection, red. È una disciplina di accodamento che gestisce la sua coda in modo intelligente. Differentemente da una disciplina di accodamento come la fifo, che scarta i pacchetti in eccedenza quando la sua coda si è riempita, la red comincia a scartare pacchetti con probabilità maggiore proporzionalmente al riempimento della coda [10].

Token Bucket Filter, tbf. È una disciplina che permette di rallentare a un valore determinato la banda di trasmissione di una interfaccia di rete. Utilizza il paradigma dei gettoni e del secchio. Consta di un generatore di gettoni, che genera n gettoni per unità di tempo, e di un secchio in grado di contenere b gettoni. Quando il secchio è pieno i gettoni vengono scartati, e quando un pacchetto di s byte deve essere trasmesso, s gettoni vengono tolti dal secchio. Se non vi sono gettoni non è possibile trasmettere pacchetti.

Stochastic Fairness Queuing, sfq. È una qdisc che cerca di distribuire equamente la possibilità di trasmettere dei dati a un numero arbitrario di flussi. Per ogni pacchetto ricevuto viene generato un hash in base all'indirizzo e alle porte di provenienza e destinazione, e per ognuno di questi hash viene creata una qdisc FIFO e il traffico viene poi trasmesso togliendo con un algoritmo round-robin dei pacchetti dalle code. A causa della possibile iniquità nell'utilizzo delle varie code, la funzione di hash viene alterata periodicamente.

3.3.2 Discipline di accodamento a classi

Class Based Queueing, cbq. Questa disciplina di accodamento implementa il modello di condivisione e gestione delle risorse di rete CBQ [9] che è stato definito nella sezione 2.3.2 di questa tesi.

Hierarchy Token Bucket, htb. Questa qdisc utilizza il concetto di token e buckets visto nella disciplina tbf, estendendolo a una struttura a classi e filtri permettendo un controllo capillare del traffico. In pratica

questa qdisc non è altro che un'altra implementazione della gestione del link in condivisione definito da CBQ.

3.3.3 Interfaccia utente

Le funzioni di controllo del traffico in Linux sono implementate all'interno del kernel per ragioni di performance, in quanto l'esportazione in spazio utente di tale meccanismo graverebbe troppo sul carico del sistema. L'interfaccia di collegamento necessaria tra il kernel e lo spazio utente è la struttura `struct tcmsg` implementata in `include/linux/rtnetlink.h`. Il client più comune che implementi questo tipo di messaggi è stato scritto da Alexey Kuznetsov e prende il nome di `tc`, traffic control.

`tc` è un classico programma utente unix a riga di comando che permette di gestire completamente il controllo del traffico di Linux, grazie a una sintassi chiara e senza fronzoli. Queste caratteristiche lo rendono però decisamente ostico da usare senza la necessaria esperienza.

La sua esecuzione visualizza i parametri di funzionamento:

```
$ tc
Usage: tc [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { qdisc | class | filter | action }
       OPTIONS := { -s[tatistics] | -d[etails] | -r[aw] | -b[atch] file }
```

Selezionando il parametro `OBJECT` è possibile gestire a seconda dei casi le operazioni sulle qdisc, sulle qdisc classful, e sui filtri.

In particolare questo tool permette di sostituire la qdisc di default associata ad una interfaccia di rete con una disciplina di accodamento o con una classe, consentendo di creare un albero di discipline con le relative priorità.

Consente inoltre di controllare tutti i parametri delle varie qdisc implementate o meno nel kernel. Normalmente infatti tutte le discipline di accodamento non ufficialmente distribuite contengono al loro interno anche l'integrazione del set di istruzioni di `tc`.

Permette di manipolare i filtri che associano dei pacchetti direttamente ad una disciplina di accodamento interna a una classe, e permette infine di restituire delle statistiche dettagliate sul funzionamento delle qdisc ricavando i dati direttamente dalle strutture interne al kernel.

Note dolenti di `tc` sono la sua carenza di documentazione, a cui cerca di sopperire il progetto collettivo Linux Advanced Routing and Traffic Control HOWTO (LARTC) [11]; e l'assoluta mancanza del sia pur minimo commento all'interno del suo codice sorgente.

3.4 Sviluppo di codice nel kernel Linux

Nei sistemi operativi moderni esiste la distinzione tra spazio utente e spazio kernel. Il ruolo di un sistema operativo è quello di fornire ai programmi utente una visione consistente dell'hardware e delle risorse del computer. Il sistema operativo (o kernel) deve verificare l'esecuzione concorrente dei programmi e proteggere il sistema da accessi non autorizzati alla memoria e alle altre risorse condivise.

Ogni processore oggigiorno fornisce delle caratteristiche hardware che permettono di rafforzare questo modo di funzionamento, limitando le operazioni che possono essere eseguite al livello utente, e permettendo solo attraverso delle funzioni predefinite - le *system calls* - l'accesso controllato al livello hardware.

In modalità kernel invece non esistono limitazioni o vincoli alla programmazione, e tutte le risorse hardware del sistema sono accessibili. Questa libertà da vincoli viene però al prezzo di non avere alcuna limitazione nemmeno in caso di errori di programmazione, che spesso risultano fatali per l'esecuzione del sistema, necessitando il riavvio della macchina.

Nella programmazione a livello utente un programma ha la possibilità inoltre di eseguire del codice che non definisce appoggiandosi a librerie esterne, mentre nella programmazione a livello kernel non si possono linkare librerie esterne, e le sole funzioni utilizzabili sono quelle esportate dal kernel stesso.

Una caratteristica utile di Linux è quella di permettere l'estensione delle funzionalità del kernel mentre questo è in esecuzione. È possibile cioè aggiungere e togliere delle caratteristiche al kernel mentre questo sta funzionando. Ogni pezzo di codice che può essere aggiunto o rimosso al kernel si chiama *modulo*, che consiste in codice oggetto (non linkato ad alcuna libreria) che può essere dinamicamente linkato a un kernel funzionante mediante il comando userspace `insmod` e scollegato mediante il comando `rmmod`.

Ogni modulo del kernel deve quindi includere le funzioni `module_init` e `module_exit` definite in `include/linux/init.h` che notificano al kernel l'inserimento del modulo e l'aggiunta di nuove funzionalità.

Nella programmazione a livello kernel particolare attenzione deve essere inoltre posta sul problema della concorrenza e delle race conditions: in una applicazione utente non multithreading infatti l'esecuzione del processo è sempre sequenziale e anche durante l'esecuzione di salti condizionati o cicli esiste una sola istanza del programma attiva.

Nella programmazione a livello kernel invece si deve sempre considerare che lo stesso segmento di codice può essere eseguito più volte contemporaneamente dal kernel. I motivi che causano questa concorrenza sono molteplici: il kernel Linux funziona per esempio su macchine multiprocessore, in cui spesso il kernel è in esecuzione contemporaneamente su più CPU nel medesimo istante.

Questo problema però non deve essere ignorato sui sistemi a singolo processore: molte periferiche sono in grado di interrompere il task corrente del processore con una richiesta di interrupt, e gli interrupt software, come ad esempio quelli causati dalla scadenza di un timer, possono arrivare in qualsiasi momento. Il kernel 2.6 su architettura monoprocessore, infine, è stato reso "preemptible", cioè interrompibile, rendendolo del tutto simile dal punto di vista della programmazione ad un kernel multiprocessore.

Come risultato, il codice scritto nel kernel deve essere in grado di funzionare in più di un contesto nel medesimo istante, e le strutture dati devono essere quindi accuratamente studiate in modo che ogni accesso a risorse condivise debba essere fornito ove necessario di meccanismi di sincronizzazione.

3.4.1 Debug del kernel Linux

Come si è già visto al punto 3.4, vi sono fondamentali differenze nella programmazione a livello kernel rispetto alla programmazione di applicazioni utente.

Tali differenze vengono esacerbate quando si tratta di trovare degli errori di programmazione - dei *bug* - all'interno di un kernel. Un kernel infatti può essere eseguito all'interno di un debugger, ma sono impossi-

bili alcune operazioni classiche di debug come l'interruzione comandata dell'esecuzione del programma e la modifica dei dati runtime.

Il kernel Linux fornisce comunque delle funzionalità che possono aiutare le operazioni di debugging, permettendo di ottenere maggiori informazioni da un kernel funzionante. Queste caratteristiche non sono normalmente abilitate sui normali kernel disponibili nelle varie distribuzioni di Linux, ma devono essere attivate sotto forma di parametri di configurazione in fase di compilazione di un nuovo kernel, in quanto implicano la creazione di un kernel di dimensioni maggiori caratterizzato normalmente da un forte degrado delle prestazioni.

Questi parametri sono disponibili soltanto dopo aver abilitato il parametro `DEBUG_KERNEL` in fase di configurazione di un nuovo kernel, che una volta attivato permette di scegliere le altre voci di configurazione delle opzioni di debug del kernel.

Le voci fondamentali da abilitare sono `CONFIG_DEBUG_INFO`, che crea un kernel completo di tutte le informazioni di debugging che potranno essere utilizzate nell'ambiente di debugging *gdb*; e `CONFIG_FRAME_POINTER` che abilita dei settaggi del compilatore `gcc` in modo da permettere il recupero da parte di un debugger delle informazioni contenute nello stack.

Una volta modificate le informazioni di configurazione del nuovo kernel è necessaria la compilazione e l'installazione del nuovo kernel. Una volta riavviato il sistema è possibile verificare il funzionamento del suo kernel mediante *gdb* attraverso il comando:

```
$ gdb /usr/src/linux/vmlinux /proc/kcore
```

dove `/usr/src/linux/vmlinux` è l'eseguibile non compresso del ker-

nel, disponibile in seguito alla compilazione dello stesso, e `/proc/kcore` è il `core` file, ovvero la copia della memoria vista da un processo, in questo caso dal kernel.

Il file `/proc/kcore` è un file fittizio, generato dal kernel, che simula le operazioni di lettura da questo file come se si trattasse di un file reale.

Questo modo di funzionamento riduce però drasticamente le operazioni che possono essere eseguite con *gdb* in quanto non è possibile alterare l'esecuzione del kernel in esecuzione e non possono essere inseriti quindi dei *breakpoints*.

Molte volte durante la programmazione dell'algoritmo si sono verificati degli errori runtime del kernel, chiamati "System Faults" o "`oops`". Nella maggior parte dei casi questi hanno causato anche il blocco del sistema, e anche nei casi fortunati in cui questo non si è verificato, dopo un segfault è comunque consigliabile riavviare il sistema in quanto non è più garantito il suo corretto funzionamento. Gli `oops` contengono però moltissime informazioni relativamente alla causa e agli errori di programmazione che l'hanno generato. In essi vengono evidenziati i valori di tutti i registri del processore e lo stack delle funzioni con i relativi parametri di chiamata.

La possibilità di utilizzare un kernel modulare è stata utilizzata da subito per ottimizzare i tempi di debugging, non richiedendo il riavvio del sistema durante le varie fasi di sviluppo dell'algoritmo.

Questa comodità viene però a scapito di alcuni settaggi aggiuntivi necessari per permettere il debug dei moduli caricati runtime: si rende necessario istruire il debugger a caricare i segmenti aggiuntivi del modulo stesso relativi alle informazioni di debug.

I moduli del kernel hanno lo stesso formato normali eseguibili, l'ELF[1], e come tali sono divisi in sezioni. Tali sezioni sono visibili come file con nome preceduto da un punto nella directory `/sys/module/nome_modulo/sections/`. Ogni modulo tipicamente contiene una dozzina di sezioni, ma quelle che interessano il debugging sono soltanto tre:

`.text`

Questa sezione contiene il codice eseguibile del modulo. Il debugger deve conoscere questo segmento per essere in grado di capire a che indirizzo si trovano le funzioni definite in modo da poter settare dei *breakpoint* e poter ricavare informazioni sul *backtrace* del kernel.

`.bss` e `.data`

Queste sezioni contengono le variabili definite nel modulo. Se una variabile viene inizializzata in fase di compilazione del modulo il suo valore va nel segmento `.data`, se questa inizializzazione non c'è e avviene runtime, il suo valore sarà contenuto nel segmento `.bss`.

Queste informazioni sui segmenti devono essere fornite a *gdb* mediante il comando `add_symbol_file` che prende come parametri il nome del modulo e poi l'indirizzo esadecimale del segmento `.text` che deve essere ricavato dalla directory `/sys/module/nome_modulo/sections/`. Per ottenere questa funzionalità si è resa necessaria la modifica di uno script preesistente [12] che permettesse di automatizzare questo comando.

[1] Executable Linkable Format

3.4.2 User Mode Linux

Le forti limitazioni imposte all'utilizzo di *gdb* appena citate hanno richiesto una analisi delle tecnologie alternative che permettessero di *debuggare* un kernel in esecuzione. Si è utilizzato User Mode Linux (UML), una versione del kernel di Linux che funziona in userspace e che deve essere eseguito all'interno di un sistema Linux funzionante.

Utilizzando UML si ottiene il risultato di avere un normale programma utente che funziona come un sistema reale, in quanto tutte le richieste all'hardware vengono dirottate a richieste emulate da UML.

I vantaggi di questo approccio sono notevoli: è possibile avere più istanze UML funzionanti contemporaneamente nella stessa macchina ospitante, condividendo addirittura le immagini dei dischi virtuali, ed è possibile creare e configurare una nuova macchina virtuale in pochissimi secondi digitando pochi comandi UNIX. Attraverso le opzioni di networking è inoltre possibile configurare e utilizzare reti di macchine virtuali di media complessità.

Il kernel UML è un normale programma utente e di conseguenza può essere eseguito quasi senza limitazioni all'interno di un normale debugger e l'occorrenza di un *crash* di sistema si risolve semplicemente riavviando un eseguibile. È inoltre possibile eseguire delle analisi *post mortem* del kernel dopo il crash, permettendo di accedere allo stack delle funzioni chiamate e di fatto riducendo di molto i tempi di debugging.

Il kernel UML essendo un programma userspace non può entrare in contatto diretto con l'hardware e quindi tutte le sue chiamate devono essere tradotte in chiamate di sistema verso il kernel ospitante. Questa limi-

tazione non é stata però di ostacolo allo sviluppo dell'algoritmo in quanto numerose periferiche, tra cui le schede di rete, sono comunque presenti in forma emulata.

L'implementazione del controllo del traffico inoltre si trova a un livello di astrazione più alto della gestione delle periferiche di rete all'interno del kernel, e l'utilizzo di queste periferiche emulate non ha creato alcuna problematica durante la fase di sviluppo dell'algoritmo.

Normali programmi utente possono essere eseguiti dal kernel UML che quindi diventa un computer virtuale con i suoi processi, le sue interfaccie di rete emulate con associati gli indirizzi ip e i suoi dischi emulati. Questa facilità e comodità di creare reti virtuali ha fatto sì che tutta la fase di sviluppo dell'algoritmo potesse esser eseguita su macchine emulate UML.

Il kernel UML ha a disposizione varie tipologie di reti emulate con le quali interconnettere le singole virtuali. Le due che sono state provate sono lo switch daemon e TUN/TAP. La prima si appoggia a un eseguibile che crea dei socket su disco e che replica le informazioni scritte verso tutte le macchine che hanno aperto quel socket, la seconda si appoggia invece a TUN/TAP che è un metodo per interconnettere la trasmissione di pacchetti con un programma in userspace.

Capitolo 4

Enhanced TCP ACK Pacing: requisiti e implementazione

In questo capitolo verrà richiamato brevemente l'algoritmo teorico ETAP definito in [2] in modo da rendere accessibile il principio di funzionamento su cui si basa l'implementazione proposta.

Di seguito verranno determinati i requisiti software e hardware e verranno definite le specifiche di funzionamento dell'implementazione. Verrà poi illustrata e commentata l'implementazione proposta in questa tesi dell'algoritmo ETAP, e sarranno poi analizzate le le problematiche tecniche e le soluzioni applicate.

4.1 Definizione di ETAP

In questa sezione si riassume [2] in modo da rendere più chiara la comprensione dell'algoritmo implementato.

4.1.1 Principio per la riduzione della banda allocata

Come si è visto nei capitoli precedenti, la quantità di nuovi dati che vengono immessi nella rete dal ricevitore sono determinati dal valore della finestra di congestione *cwnd*. Il ritmo dell'immissione di nuovi pacchetti pacchetti nella rete è determinato dall'ACK di ritorno che riconosce al mittente nuovi dati ricevuti.

Quando il mittente riceve un ACK di risposta che riconosce nuovi dati viene reintrodotta nella rete una quantità di dati pari alla dimensione dei dati riconosciuti dall'ACK ricevuto, sommata eventualmente all'incremento della *cwnd* dovuta alla fase dell'algoritmo delle congestioni in cui si trova il mittente.

La banda utilizzata dal mittente del flusso TCP è quindi influenzata direttamente dal ritmo degli ACK di risposta. A sua volta, la quantità di dati riconosciuta dagli ACK dipende dalla capacità del canale. Il mittente continua ad aumentare la quantità di dati immessi nella rete fino a quando la capacità del canale di trasmissione non si esaurisce e comincia la perdita dei pacchetti nei router intermedi.

Si consideri a questo punto la configurazione di rete rappresentata in figura 4.1 che rappresenta un trasmettitore e un ricevitore di un flusso TCP connessi a rete a larga banda, che a loro volta sono connesse a un collegamento di capacità inferiore che definiamo *collo di bottiglia*.

La figura 4.2, tratta da [5], rappresenta schematicamente un link a larga banda e un collo di bottiglia presente tra il trasmettitore e il ricevitore. La dimensione orizzontale rappresenta il tempo, quella verticale la banda. I rettangoli ombreggiati rappresentano i pacchetti e la loro dimensione P è

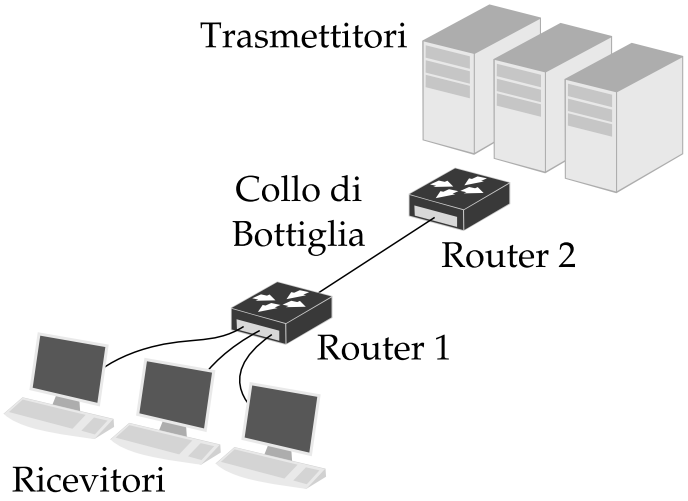

Figura 4.1: Esempio di configurazione di rete

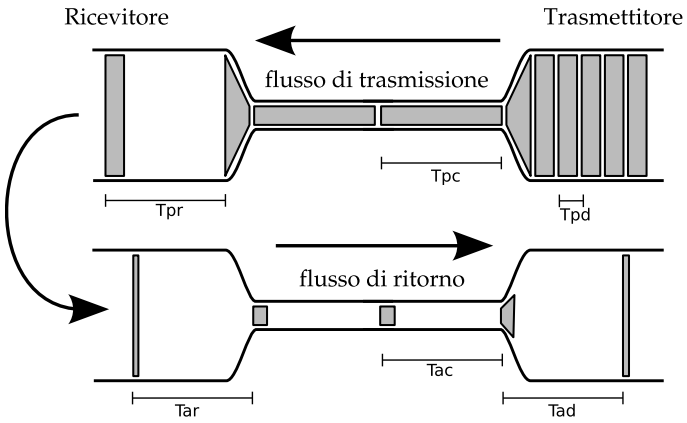

Figura 4.2: Bande e ritardi di trasmissione

data quindi da

$$P\left[byte\right] = Area\left[byte\right] = \frac{Banda\left[\frac{bit}{sec}\right]}{8} \cdot Tempo\left[sec\right]$$

P rappresenta la dimensione in *byte* del pacchetto, che coincide con la somma delle dimensioni della parte dati, D, e degli header IP e TCP, H.

Essendo P costante, quando un pacchetto viene inoltrato attraverso il collegamente più lento avrà una occupazione temporale maggiore rispetto ad un link più veloce. Sia Tp_c la spaziatura minima tra pacchetti nel collo di bottiglia.

Una volta che i pacchetti hanno superato il collo di bottiglia la loro spaziatura temporale non cambia e quindi $Tp_r = Tp_c$. La spaziatura tra gli ACK generati dal ricevitore sarà quindi $Ta_r = N \cdot Tp_r = N \cdot Tp_c$, dove N rappresenta il numero di pacchetti dati riconosciuti da ogni ACK, dove tipicamente $N = 1$ oppure $N = 2$ come richiesto da [4] (nella figura 4.2 il valore di N è stato posto a 1).

Supponendo che le dimensioni del pacchetto ACK di ritorno siano molto minori di quelle del pacchetto che riconosce, e che la capacità del canale di ritorno non sia troppo inferiore rispetto a quello di andata, si deduce che la spaziatura Ta_c sarà preservata, cioè che $Ta_c = Ta_r = Ta_d$. La quantità di dati riconosciuta dagli ACK nell'unità di tempo, che corrisponde alla banda rilevata a livello applicativo, è quindi pari a:

$$B_{riconosciuta} = \frac{N \cdot D}{Ta_d} \cdot 8 = \frac{D}{Tp_c} \quad \left[\frac{bit}{sec}\right] \tag{4.1}$$

In condizioni normali, la spaziatura temporale Tp_c è determinata dalla banda disponibile nel collo di bottiglia, cioè:

$$Tp_c = \frac{P \cdot 8}{B_{bottleneck}}$$

Sostituendo questa equazione nella 4.1 si può constatare che, a meno della dimensione dell'header del pacchetto, la banda riconosciuta dagli ACK di ritorno corrisponde effettivamente alla banda del link collo di bottiglia:

$$B_{riconosciuta} = \frac{D}{P} \cdot B_{bottleneck} \tag{4.2}$$

Trattenendo artificialmente gli ACK inviati dal ricevitore che riconoscono al mittente nuovi segmenti ricevuti correttamente, si ottiene la diminuzione della banda riconosciuta, e quindi indirettamente anche della banda utilizzata dal flusso dati.

Dopo la fase iniziale di riduzione della banda, il trasmettitore incrementerà però la sua finestra di congestione (coerentemente con la fase di congestion avoidance in cui si trova) aumentando la *flight size*, ovvero la quantità di dati non riconosciuti dal ricevitore che si trovano ancora nella rete.

Questa fase terminerà quando in uno dei router a monte del collo di bottiglia verranno scartati alcuni pacchetti per il raggiungimento della capacità massima della coda di tale router. Questa perdita di pacchetti verrà notificata al trasmettitore che dedurrà che la rete sia congestionata, e ridurrà di conseguenza la sua *cwnd*, riducendo quindi la banda utilizzata nel collegamento di trasmissione.

4.1.2 Allocazione della banda

Una volta compreso il principio di riduzione della banda utilizzata da un flusso, rimane da determinare il ritardo temporale da assegnare al sin-

golo ACK di ritorno in modo da forzare la banda occupata dal flusso di trasmissione.

Dall'equazione 4.2 si ricava la banda riconosciuta al trasmettitore data la banda del collo di bottiglia. Imponendo la nuova banda allocata $B_{allocata}$ si deve modificare anche la banda di ritorno degli ACK $B_{riconosciuta\ allocata}$, in modo da rispettare:

$$B_{riconosciuta\ allocata} = \frac{D}{P} \cdot B_{allocata}$$

Per assegnare il corretto nuovo tempo di interarrivo degli ACK in funzione della banda allocata, bisogna tenere conto del numero di byte riconosciuti da ogni ACK, che tipicamente vale uno o due SMSS[1] [13].

In [2] viene presentato un algoritmo per la stima della banda riconosciuta dal trasmettitore in base all'analisi dei numeri di sequenza associati a un determinato flusso TCP.

L'algoritmo può essere implementato definendo per ogni flusso le seguenti variabili:

smss Contiene il valore del Sender Maximum Segment Size del flusso, che deve essere ricavato leggendo i pacchetti iniziali durante la fase di instaurazione della connessione TCP.

ack_max Rappresenta il massimo ACK number riconosciuto dagli ACK di ritorno precedenti a quello corrente. Il suo incremento sta a significare che il ricevitore ha riconosciuto nuovi dati trasmessi.

ack_cur Rappresenta il valore dell'ACK number nel pacchetto ACK rice-

[1] Sender Maximum Segment Size

vuto. Se è uguale a `ack_max` significa che è stato ricevuto un ACK duplicato.

br Rappresenta il numero equivalente di byte riconosciuti dall'ACK corrente ai fini del calcolo della finestra di congestione del trasmettitore.

dupack Permette di distingure tra i casi in cui questo primo segmento non duplicato sia relativo a un pacchetto dati che permetta o meno al ricevitore di coprire tutti i segmenti mancanti, mantenendo il conteggio del numero di ACK duplicati in modo da determinare se il prossimo ACK sia un ACK parziale.

L'algoritmo per la determinazione della banda riconosciuta da un pacchetto è presentato sotto e di seguito viene spiegato il suo funzionamento.

```
 1 if(ack_cur == ack_max){
 2         br = smss;
 3         dupack += 1;
 4 }else if(ack_cur > (ack_max + dupack * smss)){
 5         br = ack_cur - dupack * smss - ack_max;
 6         if (br < 0)
 7                 br = 0;
 8         dupack = 0;
 9 }else if(ack_cur > ack_max){
10         br = smss;
11         dupack -= (ack_cur - ack_max) / smss - 1;
12 }
```

```
13 ack_max = ack_cur;
```

Linee 1-3 Quando il nuovo pacchetto ACK ricevuto non riconsce nuovi dati, cioè quando il suo ack_cur coincide con l'ack_max del flusso, significa che al ricevitore è giunto un pacchetto fuori sequenza. Lo standard [7] per il controllo delle congestioni prevede infatti l'immediata notifica al trasmettitore di ogni segmento che riconosca nuovi dati, mediante l'invio di un pacchetto ACK che non riconosca nuovi dati.

In questo caso la dimensione dei dati riconosciuti dal ricevitore viene posta a uno SMSS, e il contatore degli ACK duplicati viene incrementato per tenere traccia dell'arrivo del pacchetto al fine di calcolare successivamente il numero di segmenti riconosciuti dagli ACK successivi.

Linee 4-8 Quando il nuovo pacchetto ACK ricevuto va a riconoscere dei nuovi dati comprensivi anche dei segmenti ricevuti fuori sequenza dal ricevitore, la dimensione riconosciuta viene uguagliata alla dimensione dei nuovi segmenti riconosciuti sottratta alla dimensione degli eventuali pacchetti giunti precedentemente fuori ordine, la cui dimensione è già stata considerata.

Linee 9-12 Diversamente, nel caso il pacchetto ricevuto riconosca nuovi dati ma non vada a ricoprire completamente anche la dimensione stimata in seguito a dei pacchetti giunti fuori ordine, si considera ancora una dimensione riconosciuta pari a uno SMSS e viene decrementato il contatore dei pacchetti fuori sequenza.

Linea 13 In ogni caso viene aggiornato il valore dell'ACK massimo ricevuto, in modo da essere coerente alla prossima iterazione dell'algoritmo con la dimensione.

Una volta determinato la dimensione `br` della banda riconosciuta è sufficente forzare il ritardo del pacchetto ACK a:

$$T_{ETAP} = \frac{D}{B_{allocata}}$$

per limitare il flusso considerato alla banda desiderata.

4.1.3 Utilizzo di ETAP

L'algoritmo ETAP consiste nel trattenere nel router i pacchetti di ACK di ritorno inviati dal ricevitore per un intervallo di tempo sufficiente a indurre il trasmettitore a inferire che la linea di trasmissione sia satura. Il ritardo temporale da assegnare all'ACK è determinato in base alle informazioni contenute nell'ACK stesso e negli ACK precedenti già inviati. È quindi necessario che per ogni flusso TCP su cui viene applicato ETAP vengano mantenute tutte le variabili utilizzate dall'algoritmo viste in precedenza.

L'algoritmo deve inoltre poter accodare i pacchetti appartenenti al medesimo flusso in modo da garantire l'invio di un solo ACK alla volta e deve garantire che non avvenga alcun riordinamento dell'ordine dei pacchetti che devono essere inviati.

ETAP non richiede che il router agisca in alcun modo sul flusso dati di andata, ma limita il suo intervento alla gestione dei pacchetti ACK di ritorno. I pacchetti appartenenti a flussi TCP non controllati da ETAP, e in generale tutti i pacchetti non TCP, non devono quindi essere ritardati o

alterati dall'algoritmo, che deve di conseguenza discriminare per flusso di appartenenza e applicare il ritardo solo ai flussi controllati.

La necessità dell'algoritmo di discriminare i pacchetti in base al flusso di appartenenza obbliga particolari accorgimenti nel caso si voglia assegnare una determinata banda aggregata a una classe di flussi. Sarà necessario infatti assegnare una frazione equa della banda totale allocata ai singoli flussi e sarà necessario riassegnare tale frazione ogni qualvolta il numero di flussi associati a tale classe venga modificato.

Deve essere mantenuto di conseguenza uno stretto controllo per identificare quali flussi siano attivi al momento e quali siano invece terminati, analizzando quindi lo stato del flusso TCP come specificato in [4].

4.2 Specifica dei requisiti di ETAP

Dall'analisi delle caratteristiche di utilizzo di ETAP si possono derivare i requisiti software necessari alla sua corretta implementazione, che verranno brevemente analizzati.

Architettura e portabilità L'algoritmo ETAP deve essere implementato nel sistema operativo Linux, e deve essere indipendente dall'architettura e dall'endianness[2] del sistema. Si deve inoltre cercare di restare per quanto più possibile aderenti agli standard di programmazione, alle strutture dati e alle facility offerte dal kernel Linux.

Discriminazione dei pacchetti L'implementazione deve poter accedere agli header sia IP che TCP dei pacchetti in ingresso, e deve poter determi-

[2]Si indica con endianness l'ordine in cui i dati multiparola sono rappresentati dal sistema.

nare in base al flusso di appartenenza se il pacchetto debba essere ritardato con l'algoritmo ETAP oppure semplicemente lasciato passare senza alcun ritardo. Tutti i pacchetti appartenenti a protocolli diversi dal TCP non devono essere ritardati.

Accodamento per flusso dei pacchetti Una volta determinato il flusso TCP di appartenenza di un pacchetto ACK, si deve accodare tale pacchetto agli altri già presenti, e procedere al calcolo del ritardo da assegnare a tale pacchetto. È fondamentale che la coda sia gestita in maniera FIFO[3] per evitare che venga alterato l'ordinamento sequenziale dei pacchetti di un determinato flusso.

Manipolazione code dei pacchetti Deve essere possibile aggiungere e togliere pacchetti dalle code, e deve essere possibile spostare un pacchetto da una coda all'altra; in particolare è necessario creare una coda dei pacchetti pronti per essere inviati al mittente, a cui è già stato applicato il ritardo dall'algoritmo. In nessun caso si deve scartare un pacchetto dalla coda, o cambiare l'ordine sequenziale di pacchetti appartenenti allo stesso flusso. È possibile cambiare l'ordine di pacchetti appartenenti a code diverse, a patto però che venga rispettato il requisito precedente.

Gestione ritardo di una coda di pacchetti Si deve avere la possibilità di ritardare i pacchetti appartenenti ad una coda di una quantità di tempo variabile a seconda dei parametri della connessione in questione. La durata del ritardo non deve essere influenzata direttamente dai ritar-

[3]First-In First-Out: una coda dove il primo elemento aggiunto alla coda sarà anche il primo ad essere tolto

di delle altre code, ma solo dal gestore di allocazione di banda. Una volta che il pacchetto è stato ritardato deve essere posto in una coda senza ritardo e deve essere instradato verso il mittente al più presto possibile.

Requisiti hardware Si richiede che l'algoritmo funzioni su un processore a 400Mhz di classe intel Pentium e con una quantità di RAM non superiore ai 128MByte. Questi requisiti di sistema massimi servono a garantire che l'implementazione possa funzionare su sistemi a bassa potenza computazionale e che non richieda notevoli quantità di memoria in modo da non impedirne una implementazione su hardware embedded.

Requisiti Prestazionali In questa fase iniziale della implementazione si vuole testare il funzionamento dell'algoritmo con un numero ridotto di flussi, nell'ordine dei 50 - 100. L'implementazione deve poter allocare una determinata banda con un errore medio di allocazione non superiore al 5%. La capacità di allocazione inoltre non deve dipendere dalle bande allocate ad altri flussi in condizioni di linea di trasmissione non satura.

4.3 Implementazione di ETAP

L'algoritmo è stato implementato e testato su varie versioni del kernel Linux comprese tra la 2.6.8 e la 2.6.12. Nella versione 2.6.13 (non ancora ufficialmente rilasciata alla stesura di questa tesi) sono state inserite delle lievi modifiche alla gestione del controllo del traffico, e non è quindi garantito il funzionamento dell'algoritmo in questa e nelle successive versioni.

La patch contenente l'implementazione dell'algoritmo ETAP è stata rilasciata sotto la licenza GNU Public License ed è disponibile all'indirizzo `http://www.circolab.net/~marcogh/software/kernel/`.

Allo stesso indirizzo è disponibile anche la patch da applicare al programma `tc` per permettere al sistema di usufruire dell'algoritmo implementato. Le modifiche effettuate a tale programma sono triviali e non verranno analizzate.

Si suppone infine che l'algoritmo possa essere implementato anche in un kernel 2.4 in quanto non vi sono sostanziali differenze dall'implementazione del controllo del traffico del kernel 2.6, ma non è stata eseguita nessuna prova per confermare questa ipotesi.

4.3.1 Il controllo del traffico nel kernel

Come si è visto al punto 3.3, il kernel Linux implementa nativamente alcune discipline di controllo del traffico di rete, e la prima fase logica dell'implementazione di ETAP è stata quindi l'analisi e la comprensione delle strutture dati e dei metodi di funzionamento delle discipline preesistenti.

Come si è visto, il componente concettuale fondamentale dell'implementazione del controllo del traffico è la *disciplina di accodamento* (`qdisc`) che grazie alla sua struttura modulare permette sia di essere associata direttamente alla interfaccia di rete, e sia di essere impilata ad altre qdisc. La sua struttura diventa quindi il punto di riferimento per ogni implementazione di un nuovo algoritmo di controllo del traffico.

Ogni qdisc viene identificata internamente dal kernel con un numero a 32 bit - univoco per interfaccia di rete - rappresentato come unione di due parti da 16 bit cadauna, la maggiore *major* e la minore *minor*. Questi due

numeri consentono di identificare univocamente una disciplina di accodamento e sono utilizzati mediante il comando utente tc per creare e manipolare delle qdisc a classi e per aggiungere e togliere filtri dalla struttura stessa.

La creazione di una nuova disciplina di accodamento si ottiene mandando un messaggio al kernel attraverso la struttura `tcmsg` definita in `include/linux/rtnetlink.h` contenente nel campo `tcm_parent` il numero identificativo della disciplina cui la nuova qdisc verrà impilata, e nel campo `tcm_handle` il numero identificativo della nuova disciplina da creare.

La nuova struttura inserita deve fornire delle funzioni di callback che specifichino il funzionamento della nuova `qdisc` e che verranno richiamate quando la disciplina inizierà ad essere utilizzata. Tali funzioni sono specificate nella `struct Qdisc_ops` definita in `include/net/pkt_sched.h`.

Per una chiara esposizione dell'implementazione dell'algoritmo ETAP si necessita la comprensione di alcune di queste funzioni di callback. Per una documentazione esaustiva contenente l'elenco e la descrizione completa delle funzioni definibili si veda [14] oppure si consulti la struttura `struct Qdisc_ops` definita in `include/net/pkt_sched.h`.

enqueue accoda un pacchetto nella `qdisc`. Questa procedura viene richiamata quando un nuovo pacchetto è pronto per essere passato dal kernel all'interfaccia di rete. Tale pacchetto viene passato come parametro alla `qdisc` attraverso questa funzione, che deve ritornare la sorte assegnata al pacchetto: i valori di ritorno più comuni sono `NET_XMIT_SUCCESS` o `NET_XMIT_DROP` che corrispondono rispettivamente all'accodamento di un pacchetto nella coda di uscita o alla sua eliminazione.

Se la disciplina di accodamento ha classi, la funzione deve prima determinare la classe di appartenenza del pacchetto andando a consultare mediante la funzione `classify` la lista dei filtri associati alla `qdisc` e poi richiamare a sua volta la funzione di `enqueue` della disciplina interna selezionata e ritornare poi al kernel il valore da questa ritornato.

dequeue ritorna il prossimo pacchetto pronto per essere inviato. Questa funzione viene richiamata di solito dalle routine di controllo dell'interfaccia di rete che si rendono disponibili a inviare nuovi pacchetti attraverso la connessione, ma può essere richiamata anche da una qdisc a classi che decide di spostare un pacchetto da una coda a un'altra.

La scelta di ritornare o meno un pacchetto dipende però completamente dalla `qdisc`, che può avere dei pacchetti pronti da spedire e nonostante questo ritornare comunque il valore `NULL` che sta a significare che non vi sono pacchetti da inviare.

classify confronta il pacchetto in questione con la lista di filtri associati alla disciplina di accodamento come si è visto al punto 3.3. Ritorna la disciplina di accodamento interna a cui punta il primo filtro corrispondente. Normalmente questa funzione viene richiamata internamente alla funzione `enqueue` per associare il pacchetto alla qdisc interna corretta.

4.3.2 La gestione dei ritardi

Il kernel di Linux tiene traccia del passare del tempo nelle variabili globali `jiffies` e `jiffies_64` che vengono poste a zero quando il kernel viene avviato e vengono incrementate con un ritmo `HZ` configurabile dalle 50 alle 1000 volte al secondo. Il valore di `HZ` determina la massima risoluzione temporale utilizzabile in una gestione mediante software interrupt dei ritardi, demandando ad altre facility la gestione di ritandi inferiori a `1/HZ` secondi.

Utilizzando un `HZ` di 1000 otteniamo una risoluzione temporale di 1 millisecondo, che è conforme alle specifiche definite nel capitolo 4.2 relative al range di ritardo da applicare ad un pacchetto.

La generazione dei software interrupt viene gestita dalla struttura dati del kernel `timer_list`, definita in `include/linux/timer.h`, mediante la quale si può comandare il kernel ad eseguire una determinata funzione con un determinato parametro ad un `jiffies` definito, ovviamente localizzato nel futuro.

Si noti che essendo il valore di `HZ` dipendente da molte variabili tra cui i parametri con cui è stato compilato il kernel stesso, e l'architettura del sistema, risulta sconveniente utilizzare direttamente il valore contenuto nella variabile `jiffies` per determinare ritardi temporali. A tal proposito sono infatti presenti nel kernel delle macro, `PSCHED_US2JIFFIE()` e `PSCHED_JIFFIE2US()` che convertono il valore da microsecondi a `jiffies` e viceversa.

La struttura `timer_list` deve essere preventivamente inizializzata mediante una chiamata alla funzione `init_timer()` e poi successivamente ini-

zializzata con un puntatore alla funzione che dovrà essere eseguita alla scadenza del timer, e con un puntatore alla locazione di memoria contenente il dato da passare a tale funzione.

A questo punto ogni chiamata a `mod_timer` andrà ad attivare la funzione all'istante di tempo richiesto.

4.3.3 L'algoritmo

L'algoritmo ETAP è stato implementato nel kernel del sistema operativo Linux versione 2.6, ed è stato testato su più sottoversioni dello stesso, dalla 2.6.8 alla 2.6.12, senza riscontrare particolari differenze sia nella struttura interna del kernel sia nel comportamento nelle prove sperimentali.

Principio di funzionamento dell'algoritmo

Quando un nuovo pacchetto viene ricevuto, questo viene passato direttamente alla funzione `enqueue()` che provvede a richiamare a sua volta la funzione `classify()` che decide se si debba applicare l'algoritmo ETAP al pacchetto, andando a scandire ordinatamente tutti i filtri associati alla qdisc.

Il primo filtro che reclama l'utilizzo di tale pacchetto segnala verso quale disciplina interna quest'ultimo deve essere instradato. I filtri sono configurati in modo da determinare il flusso TCP a cui appartiene un pacchetto, e di conseguenza all'interno della disciplina sarà necessario creare una qdisc interna di tipo FIFO per ogni flusso TCP che l'algoritmo dovrà gestire, rappresentate come `queue1`, `queue2`, `queue3` in figura 4.3.

Se nessun filtro reclama l'uso del pacchetto, questo viene posto in una coda di tipo FIFO denominata `fifoq`.

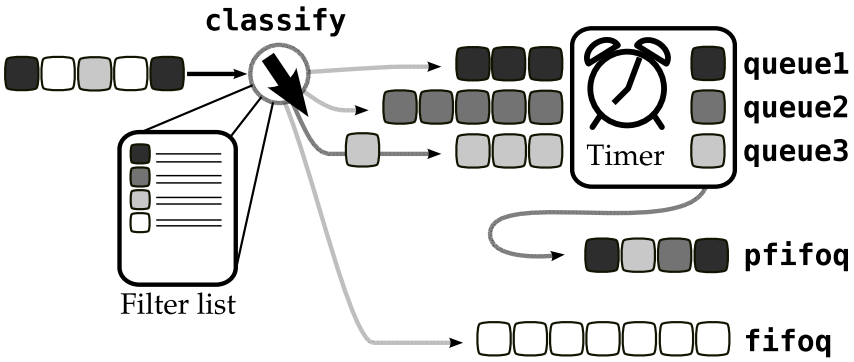

Figura 4.3: Flusso dei pacchetti nell'algoritmo

Quando vi sono pacchetti nelle code `queue1`, `queue2`, `queue3`, l'algoritmo attiva un timer per ogni singola coda, settato con il ritardo determinato nella sezione 4.1.2. Alla scadenza di questo timer viene richiamata la funzione `watchdog()` che provvede a togliere un pacchetto dalla coda in questione e ad aggiungerlo alla coda `pfifoq`.

La funzione `dequeue` ha solo il compito a questo punto di togliere i pacchetti dalle due code `pfifoq` e `fifoq` e passarle alla interfaccia di rete. Se queste due code sono vuote nessun pacchetto viene trasmesso.

Descrizione dell'algoritmo implementato

Come si è visto nel capitolo 4.3.1, il controllo del traffico in Linux è implementato attraverso delle funzioni di callback associate alla disciplina di accodamento in uso che vengono richiamate direttamente dal sistema. Le funzioni di callback `enqueue()` e `dequeue()` insieme alla funzione `watchdog()` sono sufficienti a chiarificare il funzionamento di questa implementazione, e sono presentate sotto forma di pseudocodice C.

```
1 int enqueue()
```

```
 2 {
 3         qdisc = classify(skb);
 4
 5         if(qdisc == fifoq) {
 6                 qdisc->enqueue(qdisc,skb);
 7                 return NET_XMIT_SUCCESS;
 8         }
 9
10         if(is_throttled(qdisc)) {
11                 qdisc->enqueue(qdisc,skb);
12                 return NET_XMIT_SUCCESS;
13         }
14
15         set_throttled(qdisc);
16
17         if(q->qlen == 0)
18                 pfifoq->enqueue(pfifoq,skb);
19         else
20                 queue->enqueue(queue,skb);
21
22         start_timer();
23         return NET_XMIT_SUCCESS;
24 }
25
26 struct skb dequeue()
27 {
```

```
28          skb = skb_dequeue(pfifoq);
29          if(skb)
30                  return skb;
31
32          skb = skb_dequeue(fifoq);
33          if(skb)
34                  return skb;
35
36          return NULL;
37 }
38
39 void watchdog()
40 {
41          if(queue->q.qlen = 0){
42                  unset(THROTTLED);
43                  return;
44          }else{
45                  skb = queue->dequeue(queue); /* toglie dalla queue */
46                  fifoq->enqueue(pfifoq,skb);  /* e mette in pfifoq  */
47                  start_timer();
48          }
49 }
```

`enqueue()` Come si è visto in 4.3.1 la funzione `enqueue()` viene chiamata quando arrivano dei nuovi pacchetti che devono essere instradati nella rete. Tale funzione deve ritornare il risultato dell'accodamento del pacchetto nella disciplina chiamata, cioè `NET_XMIT_SUCCESS` in caso

di accodamento riuscito.

Linee 3-8 La funzione `classify` determina in base alla lista dei filtri presenti in quale *qdisc* accodare il pacchetto ricevuto. Se il pacchetto non corrisponde alle caratteristiche richieste di alcun filtro viene accodato alla *qdisc* `fifoq` e l'esecuzione della funzione termina.

Linee 10-13 Arrivano a questo punto dell'algoritmo solo i pacchetti su cui deve essere applicato ETAP. Se la *qdisc* dove deve essere accodato il pacchetto è già limitata - *throttled*, cioè se ha già superato le limitazioni di banda a cui è sottoposta, il nuovo pacchetto viene semplicemente accodato ad essa, e l'esecuzione della funzione termina.

Linea 15 Viceversa, significa che il pacchetto appartiene a una coda che è uscita dalla fase di limitazione. L'arrivo del nuovo pacchetto quindi comanda all'algoritmo di marcare la coda come limitata: i futuri pacchetti dello stesso flusso non dovranno seguire il percorso del pacchetto corrente ma dovranno invece entrare nell'if della riga 10.

Linee 17-20 Se la coda è vuota il pacchetto viene inviato immediatamente, in quanto o si tratta del primo primo pacchetto di un flusso, oppure il traffico del flusso sta rimanendo al di sotto della limitazione imposta. Se la coda non è vuota il pacchetto viene semplicemente accodato.

Linea 22 A questo punto il pacchetto è stato inserito nella coda corretta e viene attivato il timer con il valore appena ricavato me-

diante la funzione `mod_timer()` vista in 4.3.2. Allo scadere del timer verrà attivata la funzione definita nella struttura `timer_list` precedentemente inizializzata, ovvero la funzione `watchdog()`

`dequeue()` Questa funzione viene chiamata quando il sistema è pronto per inviare un pacchetto attraverso l'interfaccia di rete verso la sua destinazione. Un solo pacchetto viene restituito per ogni chiamata di questa funzione.

Linee 28-30 Se esiste un pacchetto nella coda `pfifoq`, contenente gli ACK di ritorno che hanno già subito il ritardo temporale, questo viene inviato alla scheda di rete, e l'esecuzione della funzione termina.

Linee 32-34 Viceversa se non è presente alcun pacchetto nella `pfifoq` si passa a consegnare i pacchetti della `pfifo`.

Linea 36 Se non vi sono pacchetti in entrambe le code, non viene restituito all'interfaccia di rete alcun pacchetto. Si noti che nelle code `queue1` ...`queue3` vi possono essere dei pacchetti che non vengono restituiti in quanto le loro code sono limitate.

`watchdog()` Questa funzione viene richiamata alla scadenza del timer della struttura `timer_list`.

Linee 41-43 Se alla scadenza del timer non ci sono pacchetti da inviare è possibile togliere la marcatura di limitazione a questo flusso, che non sta superando quindi i limiti imposti.

Linee 44-46 Se nella coda si trovano ancora dei pacchetti da inviare, allora significa che il primo pacchetto da inviare ha già subito il

ritardo e quindi deve essere posto nella coda di pacchetti pronti da consegnare, quindi deve essere tolto dalla coda dove si trova e inserito nella `pfifoq`.

Linea 47 Non resta ora che riavviare il timer e terminare la funzione.

4.3.4 Problematiche relative a User Mode Linux

Come si è visto nel capitolo 3.4.2 lo sviluppo dell'algoritmo ETAP è stato eseguito quasi esclusivamente utilizzando *user mode linux* (UML), un kernel Linux compilato in modo da funzionare come programma utente. Alcune limitazioni di UML però hanno causato delle problematiche che ne hanno ridotto in alcuni casi la possibilità di utilizzo.

Reti emulate in UML

Caratteristica molto interessante di UML è la possibilità di utilizzare più macchine UML contemporaneamente implementando tra di esse delle connessionidi rete. Queste reti emulano il comportamento fisico delle connessioni di rete reali e permettono quindi la creazione di interconnessioni che vengono viste dal kernel UML in modo del tutto simile a connessioni reali. UML fornisce vari tipi di connettività emulata, che si differenziano tra loro per caratteristiche fisiche quali il livello di protocollo emulato, o il numero massimo di interfaccie di rete supportato. Le caratteristiche della configurazione di rete emulata richiesta per testare ETAP hanno limitato la scelta della tipologia di rete emulata tra due diversi tipi.

La prima modalità testata è stata quella dello *switch daemon*, che consiste nella creazione nella macchina ospitante di un socket a cui si connettono le varie macchine virtuali. Questo tipo di rete virtuale, che è stata la prima

considerata a causa della immediatezza della sua implementazione e configurazione, ha presentato da subito una forte instabilità e delle prestazioni troppo dipendenti dal carico della macchina, ed è stata di conseguenza abbandonata.

Si è quindi utilizzata la modalità TUN/TAP, che consiste nella creazione di alcuni bridge [15] a cui vengono associate delle interfaccie TUN/TAP che consentono a un programma utente di instradare e ricevere pacchetti da un'interfaccia di rete. In questo caso le interconnessioni possono essere composte associando diverse interfaccie TUN/TAP allo stesso bridge logico.

Questa modalità si è rivelata più affidabile e meno dipendente dal carico della macchina, rendendo necessaria l'implementazione dell'algoritmo su rete reale con macchine reali solo per la fase di analisi prestazionale. La modalità TUN/TAP ha infatti evidenziato nella fase di testing degli strani "riassemblamenti" dei pacchetti che invalidavano l'algoritmo per la scelta del ritardo da assegnare ad ogni pacchetto.

Separate Kernel Address Space

Nelle fasi iniziali della sperimentazione con kernel UML si è evidenziata la scarsa prestazione delle macchine emulate che ha portato a risultati poco affidabili anche nelle prestazioni di rete.

Il kernel UML normalmente funziona nella modalità definita *Tracing Thread* (TT) che è strutturata nel seguente modo; ogni processo figlio di un kernel UML è un normale processo sulla macchina ospitante, a cui viene associato un thread, il *tracing thread* appunto, che intercetta le chiamate di sistema che il processo UML vuole eseguire, e le indirizza verso la parte

superiore della memoria vista dal processo dove viene mappato il kernel UML.

Il kernel UML nella modalità TT si trova di conseguenza all'interno dello spazio di indirizzamento del processo in questione e per ovvie ragioni di sicurezza tale segmento di memoria viene quindi mappata in sola lettura durante l'esecuzione del processo, e poi rimappato in scrittura durante l'esecuzione del tracing thread. L'overhead causato dalla continua rimappatura dello spazio di indirizzamento del processo aumenta il carico del sistema e riduce le prestazioni ottenibili dalla macchina virtuale che in situazioni di carico perde complemamente responsività.

Le prestazioni di rete ottenute in situazione di carico della macchina sono risultate falsate e questa modalità di funzionamento è stata abbandonata. Il kernel UML fornisce infatti una modalità alternativa di funzionamento denominata SKAS che prevede il funzionamento del kernel UML in uno spazio di indirizzamento completamente diverso da quello dei processi UML. Tale modalità si è rivelata molto più performante e stabile e non ha risentito dei problemi di alterazione delle caratteristiche di rete che hanno caratterizzato la modalità TT.

Capitolo 5

Analisi Prestazionale

Il capitolo precedente riporta l'implementazione di ETAP che è stata svolta nell'ambito di questa tesi.

In questo capitolo verrà specificato l'ambiente in cui è stata eseguita la sperimentazione dell'algoritmo e saranno mostrate le caratteristiche e le prestazioni dell'implementazione in varie casistiche di utilizzo. Queste ultime sono state scelte con l'obiettivo di evidenziare sia le prestazioni che le limitazioni di funzionamento dell'algoritmo al fine di determinarne il campo di applicazione.

5.1 Configurazione dell'ambiente di test

L'algoritmo ETAP è stato testato in una configurazione di rete realizzata interconnettendo alcuni computer attraverso una rete locale Ethernet a 100Mbit rappresentata in figura 5.1.

L'algoritmo è stato applicato al router *R1*, dotato di un processore Intel Pentium II a 350MHz con 256MByte di RAM, su cui sono installate due

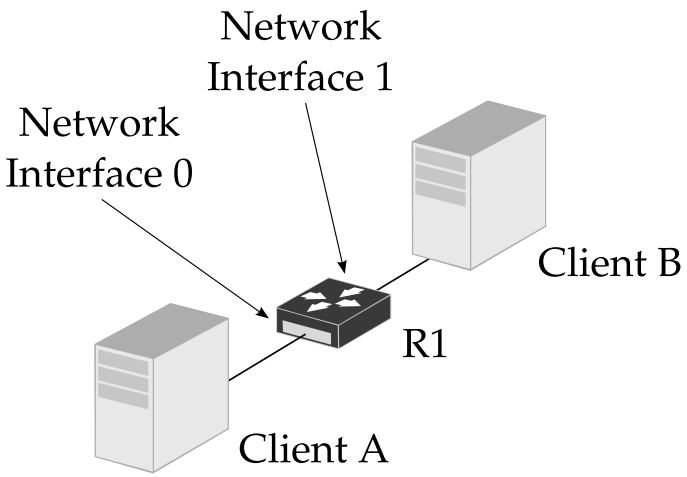

Figura 5.1: Configurazione di rete realizzata per l'analisi prestazionale dell'algoritmo ETAP.

schede di rete Ethernet equipaggiate da un chipset RTL8129.

Sul router in questione è stata installata una distribuzione Slackware 10.0 ed è stato sostituito il kernel originale con un kernel Linux vanilla[1] versione 2.6.12.5 a cui è stata applicata soltanto la patch che implementa l'algoritmo, senza accorgimenti particolari di ottimizzazione della compilazione.

Gli end-host, *Client A* e *Client B*, sono stati collegati attraverso cavi di rete a 100Mbps alle due schede di rete del router R1.

La configurazione della macchina Client B consta di un processore intel Pentium II a 400MHz e di 320MB di RAM. Il computer Client A è configurato con un processore intel Celeron a 1133MHz, ed è fornito di 384MB di RAM.

[1]Si definiscono "Vanilla" i kernel Linux ufficiali scaricati dal sito http://www.kernel.org/

Durante tutte le prove effettuate si è cercato di mantenere al minimo il carico delle macchine dovuto a compiti non collegati direttamente a questa tesi.

Per determinare la banda utilizzata sono stati approntati sulle macchine Client A e Client B delle batterie di client iperf [16] che hanno permesso di creare e monitorare singoli flussi di traffico TCP. La scelta di utilizzare iperf è stata determinata dalla particolare flessibilità di questo software, che consente due modalità di funzionamento, client e server, che permettono di implementare completamente un flusso TCP semplicemente con due istanze del programma lanciate su due host differenti.

Inoltre, le opzioni di iperf si sono rivelate molteplici e hanno permesso di specificare sia la durata del flusso - definibile come tempo o come dimensione di traffico, sia le porte TCP tra cui tale flusso viene creato.

Le informazioni fornite come output dal programma - la quantità di dati trasferita dal flusso sia per unità di tempo, sia complessivamente - sono state memorizzate integralmente e catalogate per ogni singola prova eseguita.

L'insieme dei risultati grezzi di iperf non si presta però ad una analisi statistica efficace, e si è quindi provveduto alla creazione di alcuni script in *awk* [17] che permettessero di manipolare questi risultati al fine di ottenere dei valori statistici, come il valore atteso e la deviazione standard delle misurazioni della banda associate ad una singola prova, che permettessero di ottenere grandezze significative alla stima del comportamento dell'algoritmo.

5.2 Metriche di valutazione

Per misurare le prestazioni di ETAP sono state definite delle metriche per valutarne varie caratteristiche. Le metriche definite hanno il compito di misurare quanto la banda utilizzata da ogni flusso sia corrispondente a quella allocata, e quanto sia lo scostamento della singola prova rispetto al valore medio di banda allocata ottenuto.

Errore relativo di allocazione della banda Come si è visto in 4.1.1, l'algoritmo ETAP consente di assegnare ad un flusso TCP una banda determinata analizzando le informazioni contenute nei pacchetti ACK di ritorno spediti dal ricevitore al trasmettitore.

Si noti che l'algoritmo non ricava nessuna informazione dai pacchetti che transitano sulla linea di andata del flusso, che sono quelli che contengono il traffico effettivo che deve essere limitato.

Questo parametro di valutazione rappresenta quindi l'indice di quanto la banda stimata sia corrispondente a quella effettiva. Questo errore viene misurato come errore relativo percentuale, e il suo valore ideale è 0%.

Deviazione standard della banda associata ad un flusso L'algoritmo ETAP trattiene per un tempo determinato dalle caratteristiche del flusso gli ACK di ritorno. Il ritardo associato al pacchetto può variare a causa di errori di arrotondamento o modifiche in quanto dipendente dalle caratteristiche istantanee del flusso a cui è associato.

Calcolando la deviazione standard tra le misure della banda di un flusso nei vari esperimenti eseguiti rispetto al valor medio di tali mi-

sure, si ottiene una informazione relativa al grado di affidabilità del comportamento dell'algoritmo.

5.3 Prestazioni dell'algoritmo

5.3.1 Allocazione equa della banda

La prima serie di test eseguita ha come obiettivo quello di determinare se l'algoritmo riesca ad allocare la stessa banda a una serie di flussi distinti.

I flussi, ciascuno dei quali viene monitorato attraverso iperf, vengono avviati contemporaneamente e poi lasciati funzionare per 120 secondi e in seguito interrotti. Ogni test è stato eseguito per dieci volte.

La tabella 5.1 rappresenta i risultati sperimentali ottenuti utilizzando otto flussi contemporanei a cui viene allocata la medesima banda.

Banda allocata [Kbps]	**Banda media utilizzata** [Kbps]	**Errore Relativo** [%]	**Deviazione standard** [Kbps]
10	12.421	24.21	0.44712
100	107.676	7.676	0.44311
1000	1017.701	1.7701	4.8420
10000	10426.136	4.2613	4.38409

Tabella 5.1: Allocazione equa della banda: 8 flussi contemporanei limitati alla stessa banda

L'algoritmo ha dimostrato un andamento lineare nella allocazione della banda assegnata. L'errore relativo è comunque sempre al di sotto del 7% tranne nel caso di flussi allocati a 10kbps dove la banda media ottenuta ha

uno scostamento del 24.2% dal valore assegnato. In tutte le prove il valore della deviazione standard è restato al di sotto del 4% con risultati ancora migliori per le bande superiori ai 100 Kilobit per secondo. Questo dato dimostra l'equità dell'algoritmo nell'allocare la banda ai singoli flussi, anche a fronte di uno scostamento anche importante rispetto al valore nominale assegnato.

Questo scostamento è verosimilmente imputabile ad una perdita di significato delle misure effettuate a causa dell'elevato ritardo e della conseguente perdita di pacchetti che il flusso limitato ad una banda dell'ordine dei 10 Kilobit per secondo subisce.

Limitando un flusso TCP ad una banda di 10 Kilobit per secondo implica che un ACK che riconosca 1500 byte deve essere trattenuto per circa 0.24 secondi. Il trasmettitore TCP quando si trova nella fase di slow-start incrementa il numero di pacchetti inviati a ogni round di trasmissione [1] e di conseguenza in determinate situazioni si potranno trovare alcuni pacchetti ACK in attesa di essere rilasciati dall'algoritmo. Il ritardo complessivo dell'ultimo ACK che dovrà essere inviato risulta essere la somma dei ritardi degli ACK che devono essere inviati prima di esso. Il trasmettitore TCP non vedendo ritornare l'ACK relativo a dati inviati allo scadere del Retransmission Time Out [6] provvederà a reinviare il pacchetto che considera perduto. Se quindi il tempo di ritardo complessivo di un pacchetto supera il tempo di ritrasmissione si ottiene un invio di dati già riconosciuti dal ricevitore con un conseguente spreco di banda. Di conseguenza questo algoritmo consente una buona allocazione dei flussi limitati ad una banda bande maggiore di 100 Kilobit per secondo.

5.3.2 Allocazione eterogenea della banda

Una volta determinato il comportamento dell'algoritmo nel limitare equamente dei flussi di dati, si è proceduto a analizzarne il comportamento nel caso di allocazione contemporanea a valori distinti di alcuni flussi. Si è scelto un partizionamento di una banda complessiva di 2Mbps, suddividendola in sei sottobande a cui è stato allocata nell'ordine una banda da 1000kbps, una da 500kbps, una da 300kbps e una da 200kbps. Il valore di 2 Megabit per secondo è stato scelto per evitare che i risultati del test venissero falsati da un problema di allocazione che l'implementazione dell'algoritmo ha evidenziato e che verrà evidenziato in seguito nel capitolo.

L'ordine dei flussi è stato permutato in ogni test, allo scopo di determinare l'eventuale dipendenza della banda ottenuta dall'ordine in cui si sono specificate le bande. Sono stati eseguiti in totale otto test con altrettante permutazioni delle 24 disponibili. Ogni flusso è stato mantenuto attivo per trenta secondi. Si è misurato attraverso dei client iperf il comportamento dell'algoritmo, ripetendo almeno dieci volte la misurazione per ogni permutazione.

I risultati qualitativi sono stati graficati nella figura 5.2, dove per ogni permutazione sono mostrate le bande utilizzate dai vari flussi.

La tabella 5.2 riporta i dati ottenuti relativamente alla prima permutazione testata.

Il valore della deviazione standard non ha evidenziato forti scostamenti dai valori ottenuti nelle altre classi di sperimentazioni eseguite. Si noti l'errore relativo più consistente rispetto alle altre prove eseguite, causato dal ridotto intervallo temporale in cui i singoli flussi sono stati mantenuti attivi.

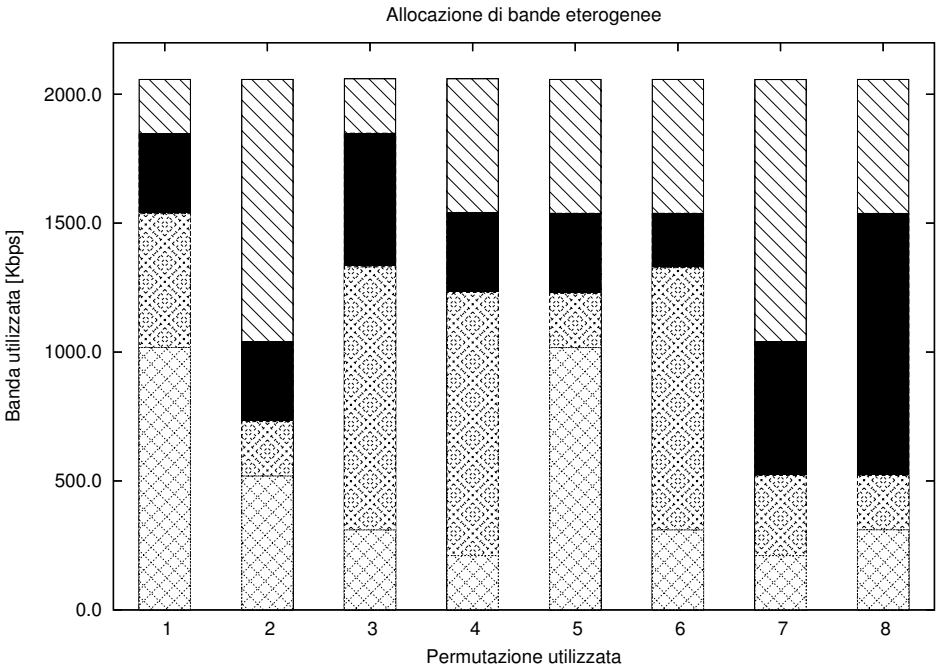

Figura 5.2: Allocazione eterogenea della banda: grafico qualitativo del comportamento dell'algoritmo con quattro flussi in otto differenti permutazioni

L'intervallo di tempo utilizzato infatti risente troppo del comportamento del flusso nel transitorio iniziale della connessione.

Banda allocata [Kbps]	**Banda media utilizzata** [Kbps]	**Errore relativo** [%]	**Deviazione standard** [Kbps]
1000	1017.646	1.764	9.0611
500	519.251	3.850	2.1843
300	310.179	3.393	0.0346
200	210.704	5.352	0.3763

Tabella 5.2: Allocazione eterogenea della banda: caratteristiche di allocazione dei flussi nella permutazione 1

Nella tabella 5.3 sono stati presentati i dati riassuntivi del comportamento dell'algoritmo nelle varie permutazioni, evidenziando solo l'intervallo di valori ottenuti nelle singole prove.

Sia la deviazione standard che la banda utilizzata nelle varie prove non si sono discostati di molto dal valore della prova considerata come campione. Sono altresì simili ad altri risultati ottenuti inducendo a concludere che non esista dipendenza dall'ordine in cui vengono allocate le singole bande.

Banda Allocata	Banda minima	Banda Massima	Deviazione standard minima	Deviazione standard massima
1000	1016.903	1020.459	8.9313	12.034
500	518.501	519.251	2.1713	2.9907
300	310.172	310.467	0.0332	0.5262
200	210.689	210.802	0.3131	0.4598

Tabella 5.3: Allocazione eterogenea della banda: tabella riassuntiva del comportamento dell'algoritmo nelle varie permutazioni

5.4 Ricerca dei limiti dell'implementazione su hw e sw disponibili

5.4.1 Massimo numero di flussi

L'obiettivo di questa classe di sperimentazioni è quello di determinare il massimo numero di flussi TCP contemporanei che l'implementazione dell'algoritmo sia in grado di supportare. Le singole bande sono state monitorate al solito utilizzando una batteria di *iperf* attivati ad istanti differenziati e mantenuti attivi contemporaneamente per almeno 100 secondi. Ciascuna prova è stata ripetuta almeno 10 volte.

Nella tabella 5.4 vengono riportati i risultati ottenuti.

Si noti che in tutte le prove eseguite non si è riuscito a determinare il massimo numero di flussi supportati dall'algoritmo in quanto tutte le sperimentazioni sono state limitate dall'eccessivo carico riportato sugli end-host che dovevano gestire le batterie di programmi *iperf* e che sono state

Numero flussi	**Banda allocata per flusso** [Kbps]	**Banda media utilizzata** [Kbps]	**Deviazione standard** [Kbps]	**Utilizzo medio CPU** [%]	**Errore relativo** [%]
40	20	24.4715	0.074118	0.128	22.3575
60	20	24.2556	0.58982	0.500	21.278
80	20	22.2071	2.3721	0.706	11.0355
40	100	106.411	0.12262	0.661	6.411
60	100	106.356	0.11130	2.023	6.356
80	100	104.056	5.8894	2.671	4.056
20	1000	1013.31	0.51081	12.202	1.331
40	1000	1013.68	0.21532	21.273	1.368
60	1000	1017.47	12.015	27.377	1.747

Tabella 5.4: Massimo numero di flussi: comportamento dell'algoritmo rispetto a bande e numero di flussi variabile

le uniche limitazioni trovate. Si noti infatti che in tutte le sperimentazioni l'utilizzo medio della CPU del router (colonna 5) è rimasto sempre al di sotto del 28%. Il valore della deviazione standard rimane abbastanza basso in tutte le prove fino a quando non si raggiungono i limiti di risorse degli end-host che di fatto causano una perdita di responsività delle macchine e in alcuni casi anche l'interruzione temporanea del traffico di rete da loro sostenuto.

Osservando le ultime righe della tabella 5.4 si può ipotizzare un andamento lineare del carico medio della CPU prevedendo un massimo di 200 flussi a 1Mbps gestibili dall'algoritmo. Purtroppo non è possibile azzardare alcuna previsione di comportamento per i flussi limitati ad altri valori a causa delle limitazioni degli end-host evidenziate prima.

5.4.2 Massima banda allocabile ad un flusso

Come si è visto precedentemente, l'implementazione dell'algoritmo ETAP di questa tesi prevede la manipolazione della coda dei pacchetti TCP di ritorno, inviati dal ricevitore al trasmettitore. Questa manipolazione avviene a diversi livelli: i pacchetti vengono accodati, ritardati e riaccodati durante il loro passaggio nel router. Queste operazioni influiscono sulle caratteristiche complessive del flusso TCP su cui è applicato ETAP.

In particolare, l'utilizzo di un ritardo con risoluzione temporale non infinitesima limita la banda massima utilizzabile da un flusso.

Scopo di questa serie di test è quella di determinare la limitazione alla massima banda allocabile ad un singolo flusso TCP. Le prove sono state eseguite utilizzando un solo flusso TCP gestito da *iperf* mantenuto attivo

per almeno 30 secondi, ripetendo ogni prova almeno 10 volte e scartando il risultato peggiore e quello migliore.

Nella tabella 5.5 vengono riportati i risultati ottenuti.

Banda allocata [Kbps]	**Banda media utilizzata** [Kbps]	**Deviazione standard** [Kbps]	**Errore relativo** [%]
10	14.345	0.03676	43.45
100	114.640	0.7326	14.64
1000	1030.68	0.008686	3.068
10000	11585.6	2.3276	15.856
100000	22693.9	1.9240	77.3061

Tabella 5.5: Massima banda allocabile ad un flusso: ricerca del limite allocabile ad un singolo flusso

L'algoritmo presenta un andamento quasi lineare nell'utilizzo della banda allocata ad un flusso. Si evidenzia un leggero scostamento già intorno ai 10Mbps che diventa poi sostanziale sui 100Mbps. Le prove eseguite evidenziano come non sia possibile allocare più di 22.6Mbps ad un singolo flusso. Si vedranno in seguito le motivazioni di queste limitazioni.

5.4.3 Massima banda di trasmissione aggregata

Obiettivo di questa serie di sperimentazioni è determinare la massima quantità di banda che l'algoritmo riesce a gestire. Le singole bande di trasmissione sono state monitorate al solito attraverso una batteria di *iperf* sugli end-host che hanno fornito le informazioni sull'utilizzo del canale di tra-

smissione dei flussi. Si è proceduto attivando inizialmente un solo flusso allocato con 10Mbps e misurandone la banda utilizzata, incrementando poi il numero di flussi fino a superare la banda massima gestibile dall'interfaccia di rete. Tutti i flussi sono stati attivati contemporaneamente e sono stati monitorati per almeno 100 secondi. Ogni esperimento è stato ripetuto per almeno 10 volte. Il grafico 5.3 contiene una rappresentazione quantitativa del comportamento dell'algoritmo.

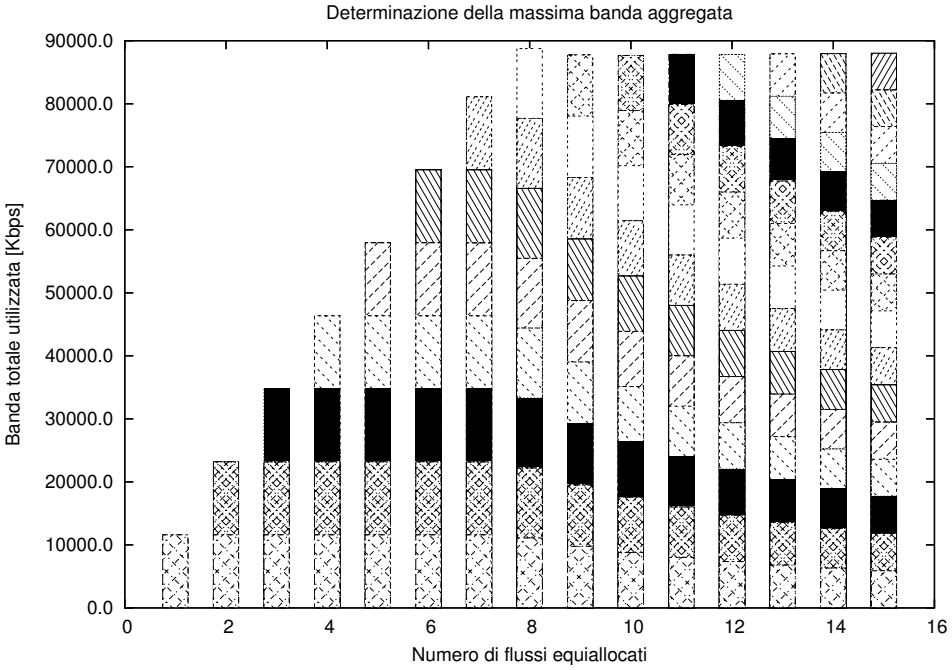

Figura 5.3: Massima banda di trasmissione aggregata: comportamento con numero variabile di flussi equiallocati

La tabella 5.6 contiene i risultati sperimentali ottenuti.

Come si può notare osservando la terza colonna, in caso di banda aggregata superiore alla banda disponibile nella linea di trasmissione si ottiene

Numero flussi	**Banda allocata media** [Kbps]	**Deviazione standard** [Kbps]	**Banda aggregata complessiva** [Kbps]
1	1158.67	1.253	1158.67
2	1158.77	1.364	2317.54
3	1158.64	1.484	3475.92
4	1158.74	1.111	4634.99
5	1158.74	1.009	5793.74
6	1158.69	1.349	6952.16
7	1158.60	3.963	8110.20
8	1109.57	95.821	8876.63
9	974.98	21.941	8774.83
10	876.57	25.295	8765.73
11	798.56	22.563	8784.22
12	731.72	29.434	8780.64
13	676.25	31.346	8791.29
14	628.23	33.212	8795.33
15	586.88	39.622	8803.26

Tabella 5.6: Massima banda di trasmissione aggregata: comportamento con numero variabile di flussi equiallocati

un incremento della deviazione standard della banda utilizzata dai flussi.

Il comportamento dei singoli flussi quando si raggiunge la limitazione del canale di trasmissione subisce un ovvia diminuzione della banda allocata al singolo flusso, visibile chiaramente nella seconda colonna. In questa situazione l'algoritmo non agisce però in modo equo tra le varie bande, come evidenzia chiaramente la colonna delle deviazioni standard. Tale valore rivela una ampia variazione delle singole bande rispetto al valore medio mostrando un sostanziale squilibrio nell'allocazione delle singole bande.

5.4.4 Variazione della banda a seconda del diverso partizionamento in flussi

Obiettivo di questa classe di test è quello di determinare il comportamento dell'algoritmo nel partizionare una determinata banda in un diverso numero di flussi. Si è fissata la banda totale di trasmissione a 10Mbps e la si è suddivisa equamente in un numero N_{flussi} di sottobande. Si è quindi variato il valore di N_{flussi} per verificare il corretto utilizzo della banda da parte dei singoli flussi, ripetendo 10 volte ogni singolo test. I flussi sono stati creati mediante delle batterie di iperf e sono stati mantenuti attivi per 30 secondi.

La tabella 5.7 contiene i risultati sperimentali ottenuti.

Si può notare come le prestazioni l'algoritmo tendano a migliorare con l'aumentare del numero di flussi in cui la banda disponibile viene partizionata. Questo comportamento è causato da un problema di allocazione alle bande superiori ai 4 Megabit per secondo circa e che verrà analizzata nella sezione seguente.

Numero flussi	Banda allocata per flusso [Kbps]	Banda utilizzata media [Kbps]	Deviazione standard [Kbps]	Banda utilizzata totale [Kbps]	Errore relativo [%]
1	10000	11577.400	0.676	11577.400	15.774
2	5000	5805.356	0.746	11610.712	16.1071
3	3333	3875.826	3.577	11627.478	16.2864
4	2500	2600.364	2.231	10401.456	4.01456
5	2000	2129.974	1.217	10649.870	6.4987
6	1666	1804.768	1.476	10828.608	8.32941
7	1428	1470.211	0.743	10291.477	2.95595
8	1250	1308.495	0.614	10467.960	4.6796
9	1111	1179.574	0.643	10616.166	6.17228
10	1000	1030.208	0.641	10302.080	3.0208

Tabella 5.7: Variazione della banda allocata a seconda del numero di flussi

5.5 Limitazioni architetturali dell'implementazione

5.5.1 Quantizzazione delle bande allocabili

Come si è visto al capitolo 4.3.2, l'implementazione dell'algoritmo ETAP sviluppato utilizza la facility *timer_list* del kernel Linux per ritardare i pacchetti di ritorno del flusso TCP controllato. L'implementazione di questo timer viene scandita dallo scorrere dei *tick* del sistema, che vanno a scadenza regolare a richiamare la lista dei timer della struttura timer_list. La frequenza di questi tick è controllata dal valore della costante `HZ` fissata in fase di compilazione che rappresenta appunto la frequenza dei tick al secondo. Il valore di questa variabile `HZ` dipende dall'architettura hardware su cui il kernel Linux è in esecuzione, e dalla versione stessa del kernel.

In particolare, nella versione 2.6.12.5 implementata su un processore intel, la costante `HZ` assume il valore di 1000, andando quindi a limitare la risoluzione temporale dei tick del sistema, e quindi anche la frequenza di scansione della struttura timer_list, a 1 millisecondo.

Si ottiene di conseguenza una discretizzazione della scala temporale degli istanti in cui possono essere inviati pacchetti, il che implica delle limitazioni sulle caratteristiche prestazionali dell'implementazione dell'algoritmo ETAP.

Il ricevitore del flusso TCP dovrebbe [13] inviare un pacchetto di ACK al trasmettitore ogni 2 pacchetti dati ricevuti correttamente. Se consideriamo una MSS del pacchetto di circa 1500 byte, potendo il nostro sistema riconoscere solo 2 pacchetti per ogni ACK e potendo inviare solo 1000 pacchetti al secondo, otteniamo una stima di banda riconosciuta massima dall'algoritmo di circa 24Mbps. Estendendo questo ragionamento ai flussi che de-

vono essere ritardati con tempi dell'ordine del millisecondo, otteniamo una quantizzazione non uniforme della banda allocabile dall'algoritmo, stimata nella tabella 5.8.

Ritardo medio [ms]	**Frequenza di emissione** [Hz]	**Banda max stimata** [Mbps]
1	1000	24.0
2	500	12.0
3	333	8.0
4	250	5.8
5	200	4.8
6	166	3.9
7	142	3.4
8	125	3.0
9	111	2.6
10	100	2.4
...	...	...

Tabella 5.8: Quantizzazione teorica della banda allocabile

Si noti che questa limitazione non dovrebbe però presentarsi con bande a cui viene allocata una banda più bassa in quanto l'errore di quantizzazione non è uniforme e diminuisce al diminuire della banda allocata.

Si è cercato quindi di determinare la fondatezza delle ipotesi precedenti creando una serie di test con l'obiettivo di misurare la banda utilizzata dall'algoritmo e lo scostamento di questo valore rispetto al valore allocato.

Si è eseguito il test allocando una alla volta tutte le bande comprese tra 1Mbps e 15Mbps con un passo di 1Mbps. Per ogni singola banda è stato utilizzato un solo flusso TCP monitorato da iperf, e ogni banda è stata testata 10 volte.

Il risultato sperimentale ottenuto è rappresentato nella figura 5.4.

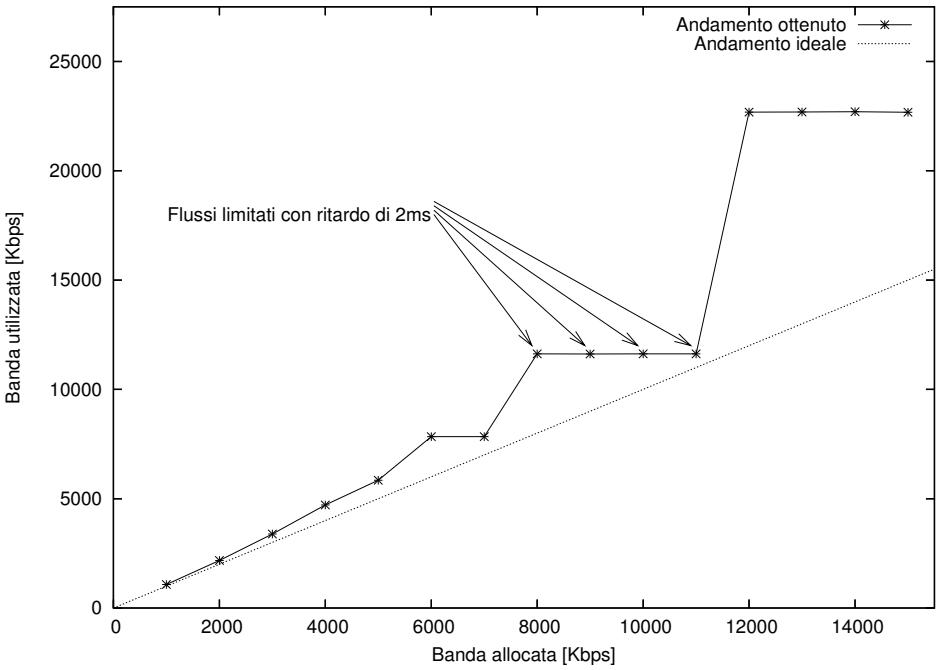

Figura 5.4: Quantizzazione della banda allocata dall'algoritmo

Dall'analisi di questa figura si evince che la quantizzazione delle bande allocabili prevista si è verificata. Si noti infatti che le bande a 7Mbps e a 8Mbps sono state allocate entrambe a 7.8Mbps circa e che tutte le bande da 8Mbps a 11Mbps estremi compresi sono state allocate ad una banda reale di circa 11.6Mbps.

Nella tabella 5.9 sono contenuti per completezza i risultati numerici del

test eseguito.

Banda allocata [Kbps]	**Banda media utilizzata** [Kbps]	**Deviazione standard** [Kbps]	**Errore relativo** [%]
1000	1076.10	6.76429	7.61
2000	2173.47	2.4102	8.6735
3000	3384.29	3.09797	12.8097
4000	4715.27	3.80694	17.8818
5000	5845.75	1.86617	16.915
6000	7841.29	1.10908	30.6882
7000	7840.89	2.13242	12.0127
8000	11622.3	11.8788	45.2787
9000	11619.1	24.4049	29.1011
10000	11625.4	8.73787	16.254
11000	11626.2	7.01811	5.69273
12000	22683.4	32.0944	89.0283
13000	22688.0	21.4022	74.5231
14000	22699.3	2.0987	62.1379
15000	22677.7	30.2835	51.1847

Tabella 5.9: Quantizzazione della banda allocata dall'algoritmo

La quantizzazione della banda teorizzata si è quindi verificata e i valori di quantizzazione sono compatibili con quelli ipotizzati. Questa limitazione non è presente nelle allocazioni a valori di banda più bassi in quanto la densità dei livelli di quantizzazione aumenta al diminuire della banda

allocata, e già negli intorni del Megabit diventa quasi trascurabile e comunque assimilabile all'errore percentuale di allocazione che l'algoritmo ha evidenziato in tutti i test.

Una implementazione futura di ETAP dovrà di conseguenza aggirare questi problemi con una diversa implementazione che in particolare non utilizzi la struttura timer_list fornita dal kernel limitata fortemente dal valore di `HZ` con cui il kernel è stato compilato. Il kernel di Linux fornisce infatti altre facility [12] per la gestione degli eventi temporali che però non sono stati considerati in questo lavoro di tesi.

Un'altra possibile soluzione potrebbe essere quella di inviare più di un pacchetto di ACK ogni volta che la funzione di `watchdog()` viene attivata.

Capitolo 6

Conclusioni e sviluppi futuri

Lo scopo di questa tesi è stato quello di determinare i requisiti e le specifiche di funzionamento dell'algoritmo ETAP, di implementare all'interno del kernel di Linux l'algoritmo, e di determinare la validità dell'implementazione svolta mediante una analisi dei risultati sperimentali ottenuti.

La letteratura propone delle metodologie per la gestione delle comunicazioni TCP implementate sui router di interconnessione tra reti diverse che agiscono sul flusso di andata della connessione TCP da controllare.

Questi metodi implementano il controllo del traffico sulla tratta dati della comunicazione TCP accodando i pacchetti diretti verso il ricevitore e rilasciandoli in modo controllato. In caso di connettività a banda di capacità elevata la dimensione occupata dalle code può diventare considerevole ed esaurire la memoria disponibile nel router. In questa situazione l'unica soluzione è quella di cominciare a scartare i pacchetti ricevuti fino a che non si sia liberata della memoria nel router.

In questa tesi è stato implementato un metodo alternativo per il controllo del traffico TCP denominato *Enhanced TCP ACK Pacing* (ETAP) definito

da Pedretti in [2] che agisce sul canale di ritorno del flusso da controllare ritardando la consegna al trasmettitore dei pacchetti di ACK che vengono ritornati dal ricevitore, ignorando completamente il canale di andata di tale flusso.

L'intervento sui pacchetti ACK inviati dal ricevitore ha permesso di diminuire la quantità di memoria utilizzata dal router per controllare un flusso di caratteristiche similari controllato con metodi classici.

Le dimensioni dei pacchetti ACK sono infatti circa del 95% più piccoli rispetto alla dimensione del pacchetto dati che riconoscono, permettendo quindi di controllare un maggiore numero di flussi a parità di memoria disponibile nel router.

L'algoritmo è stato implementato all'interno del kernel del sistema operativo Linux e le sue prestazioni sono state valutate utilizzando tale algoritmo per controllare il traffico di una rete di computer interconnessi. Dai risultati dei test sperimentali si è verificato che mediante ETAP è possibile allocare una determinata banda arbitraria a dei flussi TCP.

Si è evidenziato una buona capacità di controllare dei flussi allocati sia in modo equo che in modo eterogeneo, senza che si siano evidenziate dipendenze tra le bande utilizzate dai singoli flussi.

I risultati ottenuti lasciano inoltre intuire la capacità dell'algoritmo di gestire un elevato numero di flussi TCP contemporanei. Sono comunque necessarie altre sperimentazioni in questa direzione per determinare l'esatto limite dell'implementazione.

Si sono evidenziate però alcune situazioni di utilizzo in cui l'implementazione proposta non si è rivelata efficace nel limitare i flussi controllati.

Nel caso di allocazione complessiva dei flussi controllati ad un valore

superiore alla banda massima disponibile si è verificato un notevole incremento della deviazione standard dei flussi che indica un comportamento iniquo dell'algoritmo con i vari flussi controllati.

Nel caso di flussi limitati a una banda di 10 Kilobit per secondo, inoltre, si è ottenuto un errore relativo di allocazione del flusso pari al 24% circa. Questo errore può essere però imputato alla perdita di significato delle misurazioni effettuate. Forzando la banda utilizzata a tali valori si ottengono infatti dei ritardi da assegnare al singolo pacchetto dell'ordine del secondo. I pacchetti ACK ritardati sono inoltre accodati dall'algoritmo e questo si traduce in un ritardo complessivo che per alcuni pacchetti ACK che si trovano nella coda può superare il valore dell'RTO causando di conseguenza la ritrasmissione del segmento che il trasmettitore considera perduto.

Un possibile sviluppo futuro potrebbe indirizzarsi verso la soluzione di questo problema, andando a modificare le informazioni relative alla finestra di ricezione che il destinatario ritorna al mittente del flusso TCP come sperimentato in [1] e [18]. Il valore della finestra di congestione *cwnd* del trasmettitore infatti deve essere sempre inferiore al valore della finestra di ricezione del ricevitore *rwnd*, e questo si traduce nella possibilità di comandare il numero dei nuovi pacchetti dati che il trasmettitore introduce nella rete.

L'algoritmo implementato ha inoltre evidenziato problemi controllo di flussi allocati a valori di banda elevati a causa da una quantizzazione dei livelli allocabili. Questa quantizzazione è causata dalla discretizzazione del ritardo applicabile al singolo pacchetto che nell'implementazione eseguita è pari ai multipli interi di un millisecondo. In particolare l'imposizione di un limite di banda ad un determinato range di valori si traduce nell'appli-

cazione dello stesso ritardo, con la conseguente limitazione alla medesima banda dei flussi così controllati.

Di conseguenza una futura estensione dell'algoritmo potrebbe essere quella di ricercare una implementazione diversa della gestione dei ritardi da quella proposta in questa tesi. Il kernel di Linux [12] fornisce infatti altre facility per la gestione dei ritardi temporali che non sono state analizzate. Un'altra strada percorribile potrebbe essere quella di gestire l'invio di più pacchetti per ogni intervallo temporale.

Un altro possibile sviluppo futuro potrebbe essere quello di implementare una gestione della banda allocata a classi di flussi e non a singoli flussi come avviene adesso. Per fare questo la strada percorribile potrebbe essere quella di utilizzare le funzionalità di connection_tracking [19] offerte dal kernel Linux in modo da poter variare il ritardo da applicare a una classe di flussi a seconda del numero di flussi appartenenti ad essa.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 303–314, 1998.

[2] F. Pedretti, "Sistemi di controllo dei parametri TCP per la gestione della banda in reti geografiche," Feb. 2005. Università degli studi di Brescia, Facoltà di Ingegneria.

[3] J. Postel, "Internet Protocol," RFC 791, IETF, Sept. 1981.

[4] J. Postel, "Transmission Control Protocol," RFC 793, IETF, Sept. 1981.

[5] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pp. 314–329, ACM Press, 1988. ISBN 0-89791-279-9.

[6] V. Paxson and M. Allman, "Computing TCP's Retransmission Timer," RFC 2988, IETF, Nov. 2000.

[7] M. Allman, V. Paxson, and W. R. Stevens, "TCP Congestion Protocol," RFC 2581, IETF, Apr. 1999.

[8] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *Internetworking: Research and Experience*, vol. 1, pp. 3–26, 1990.

[9] Sally Floyd and Van Jacobson, "Link-sharing and Resource Management Models for Packet Networks," *IEEE/ACM Transactions on Networking*, vol. 3, pp. 365–386, Aug. 1995. ISSN 1063-6692.

[10] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, Aug. 1993. ISSN 1063-6692.

[11] B. Hubert, "Linux Advanced Routing and Traffic Control HOWTO." `http://lartc.org/howto/`.

[12] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, third ed., Feb. 2005. ISBN 0-596-00590-3.

[13] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122, IETF, Oct. 1989.

[14] W. Alsemberger, "Linux Network Traffic Control - Implementation Overview," in *Proceedings of 5th Annual Linux Expo, Raleigh, NC.*, pp. 153–164, May 1999.

[15] U. Böhme, "Linux Bridge STP HOWTO," Jan. 2001. `http://www.tldp.org/HOWTO/BRIDGE-STP-HOWTO/`.

[16] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf - The TCP/UPD Bandwidth Measurement Tool," May 2005. `http://dast.nlanr.net/Projects/Iperf/`.

[17] A. Aho, B. Kernighan, and P. Weinberger, *The AWK Programming Language*. Addison-Wesley, 1988. ISBN 0-201-07981-X.

[18] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, "TCP rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 45–58, 2000.

[19] R. Russel and H. Welte, "Linux netfilter Hacking HOWTO," July 2002. `http://www.netfilter.org/documentation/HOWTO/-netfilter-hacking-HOWTO.html`.