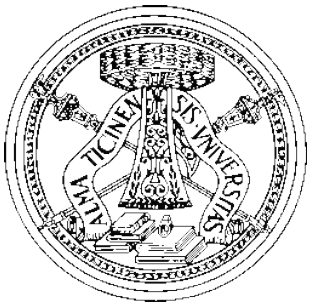

Università degli Studi di Pavia
Facoltà di Ingegneria
Sede di Mantova

Corso di Laurea Triennale in Ingegneria Informatica

Implementazione di lossy link su kernel Linux 2.6

Relatore:
Prof. Giuseppe Rossi

Tesi di laurea di:
Filippo Bistaffa

Correlatore:
Dott. Ing. Emanuele Goldoni

Anno Accademico 2008/09

Indice

Elenco delle figure

Capitolo 1

Introduzione

"La disumanità del computer sta nel fatto che, una volta programmato e messo in funzione, si comporta in maniera perfettamente onesta"

Isaac Asimov

Gli errori di trasmissione nei sistemi di comunicazione e, in particolare, la loro corretta prevenzione rappresentano una parte fondamentale dello sviluppo dei moderni metodi di ricetrasmissione. È infatti impensabile costruire un sistema che ignori completamente questo tipo di situazioni, vista la loro frequenza e la loro potenziale dannosità.

Nel contesto delle reti di calcolatori, oltre ai ben noti problemi di congestione (che tuttavia qui non tratteremo), la gestione degli errori dovuti alla perdita di pacchetti viene implementata in vari modi, concordemente con il contesto e con le funzionalità che il protocollo in questione deve supportare. Esistono sostanzialmente due approcci, che si traducono in altrettanti protocolli che affrontano in maniera alternativa il problema nello stack di rete TCP/IP, alla base della stessa rete Internet.

Il protocollo TCP (Transmission Control Protocol) garantisce la totale integrità del trasferimento e quindi adotta una rigida politica riguardo gli errori. Ogni pacchetto deve necessariamente arrivare al ricevente (che lo deve confermare), altrimenti il problema viene rilevato e corretto, provvedendo a ritrasmettere i dati finché questi non vengono recapitati con successo.

Totalmente diverso è l'approccio adottato dall'UDP (User Datagram Protocol), che garantisce una gestione più snella del trasferimento, riducendo l'overhead nella trasmissione; ciò lo rende adatto alle applicazioni nelle quali la reattività e le basse latenze sono requisiti imprescindibili. Ovviamente queste caratteristiche si ripercuotono sulla gestione degli errori, che sono sostanzialmente ignorati.

È quindi fondamentale avere a disposizione un set di strumenti che permetta di generare, in maniera controllata, una perdita di pacchetti su uno o più link di rete e affinare la progettazione in accordo con i risultati ottenuti.

L'oggetto di questa tesi è esporre lo sviluppo di un particolare device driver denominato *Insane*, che permette di realizzare un link di trasmissione affetto da perdita controllata, ovvero secondo modelli selezionabili dall'utente.

Gli obiettivi preposti per lo sviluppo di questo software sono innanzitutto l'elevata portabilità, ovvero la possibilità di distribuire e rendere operativo il driver in maniera semplice e veloce. Essendo destinato all'installazione su nodi di rete intermedi, dotati quindi di un'enorme variabilità hardware e software, è impensabile realizzare una soluzione che dipenda dall'architettura delle interfacce di rete: questo approccio implicherebbe infatti la modifica del codice sorgente dei driver, con successiva ricompilazione, per ogni singola macchina. Si è quindi optato per il collaudato approccio "a layer", già ampiamente utilizzato per la realizzazione dello stack ISO/OSI, nel quale ogni strato è realizzato in modo da offrire un'interfaccia standard sia in entrata che in uscita, permettendo ad ogni livello di rendersi indipendente ed al tempo stesso intercambiabile con altri.

Altro requisito essenziale per garantire una buona usabilità del driver è una elevata facilità di configurazione, che deve avvenire in tempo reale e senza la necessità di eseguire ricompilazioni di codice. Anche sotto questo punto di vista è stata rispettata la struttura adottata dalla maggior parte dei software in questo ambito, che prevede un programma implementato da un modulo interfacciato direttamente con il sistema operativo, che realizza di fatto le funzioni fornite dal driver. La struttura modulare tipica di Linux è risultata molto comoda in fase di sviluppo in quanto ha permesso di correggere in maniera agile e veloce i bug via via riscontrati, eseguendo la rimozione e il caricamento del modulo una volta sistemato il problema, senza quindi rendere necessaria l'intera ricompilazione del kernel.

La configurazione del suddetto componente avviene invece con un programma userspace (denominato *Insanely*), che permette di variare i parametri e quindi il comportamento del device driver a *runtime*.

Lo sviluppo di tale driver è partito dal lavoro messo a disposizione da Alessandro Rubini, che aveva improntato il lavoro sul kernel Linux 2.4, al giorno d'oggi obsoleto. Il primo passo dello sviluppo è stato quindi il porting del codice alla versione attuale 2.6, operazione che ha richiesto una notevole quantità di lavoro a causa della sostanziale riscrittura delle interfacce software alle quali i network driver si appoggiano e la correzione di alcuni bug presenti nel codice originale.

A livello implementativo, la struttura del driver è quella di un normale Linux network driver, alleggerita dal fatto che la reale trasmissione è delegata al device a cui Insane si appoggia. Internamente al driver, le operazioni che vengono eseguite sono la realizzazione della perdita tramite i modelli precedentemente illustrati, che determinano se il pacchetto in esame deve essere o meno inoltrato. In caso di esito positivo, il pacchetto viene trasferito dal buffer di Insane a quello dell'interfaccia reale che si occupa della trasmissione, altrimenti viene semplicemente scartato.

Insane offre due modelli di perdita dei pacchetti: è possibile scegliere di inoltrare una percentuale prefissata del traffico di rete in transito, ideale per simulare un canale rumoroso nel quale, in maniera casuale, un numero pressoché costante di frame ogni cento viene perduto. Inoltre è disponibile una modalità a tempo, con la quale vengono alternativamente scartati e trasmessi pacchetti per intervalli temporali selezionabili dall'utente.

Una volta terminato lo sviluppo e l'installazione del driver, si è proceduto con alcuni test per verificare che il comportamento e le prestazioni di Insane fossero comparabili con quelle offerte da alcuni strumenti software già presenti sui sistemi GNU/Linux, quali Iptables e Netem.

Iptables è un programma userspace che permette di avere accesso e configurare le tabelle e le regole fornite dal firewall del sistema operativo, agendo sul framework offerto da Netfilter. Esso consiste in una serie di *hook*, ovvero dei checkpoint presenti in vari punti dello stack di rete, che vengono richiamati man mano che un pacchetto attraversa i vari strati protocollari. Se una determinata applicazione è registrata presso il framework di Netfilter, viene notificata della presenza del pacchetto, e ottiene l'autorizzazione per esaminarlo, eventualmente inoltrarlo, oppure scartarlo.

Iptables è altamente configurabile ed espandibile, grazie all'elevato numero di estensioni che permettono di modificarne a piacimento il comportamento. In particolare, per implementare la perdita casuale di pacchetti secondo una certa percentuale, è stato usata l'estensione *statistic*, che riunisce al proprio interno tutte le funzioni di tipo statistico, appunto. Queste qualità di Iptables hanno permesso di realizzare facilmente un comportamento comparabile ad Insane, offrendo un buon punto di riferimento per valutare l'effettiva bontà del driver, che si è rivelato equivalente e in alcuni punti addirittura superiore alla controparte.

Come secondo termine di paragone è stato usato un particolare modulo kernel fornito dal sistema operativo denominato *Netem*, che sta per Network Emulation. Esso infatti, a differenza di Iptables, più improntato al controllo e alla gestione dei pacchetti che transitano su una determinata macchina, è usato per emulare il comportamento e le proprietà delle reti WAN, ovvero quelle di notevole estensione, caratterizzate da alti valori di ritardo nei pacchetti oltre che perdita, duplicazione e riordino degli stessi. Netem, facendo parte del kernel, viene fornito in tutte le distribuzioni GNU/Linux, e può essere configurato dall'utente tramite il tool a riga di comando `tc` presente nel pacchetto *iproute2*, che svolge il ruolo di Iptables per Netfilter.

I test hanno evidenziato come Insane faccia registrare prestazioni in linea con Netem, mostrando una buona precisione nella simulazione della perdita controllata. Inoltre, dal punto di vista dell'overhead computazionale, il software ha richieste molto esigue, influendo in maniera pressoché nulla sul carico della macchina, il che porta a concludere che possa essere realmente impiegato con le finalità sinora esposte.

Per poter meglio spiegare quanto brevemente indicato in questa introduzione, la presente tesi è strutturata come segue. Dopo una breve introduzione alle reti a commutazione di pacchetto, trattate nel Capitolo 2, verrà analizzato nel dettaglio il fenomeno del packet loss nel Capitolo 3, con un'analisi delle soluzioni disponibili e dei software di test. Nel Capitolo 4 sarà proposto in maniera dettagliata il lavoro di sviluppo svolto sul driver Insane, esponendone le caratteristiche ed i cambiamenti apportati. Il successivo Capitolo 5 illustrerà i test comparativi effettuati fra Insane e i due software di riferimento, Iptables e Netem, per mezzo di tool standard come *Iperf* e *Ping*. Alcune riflessioni conclusive sul lavoro svolto e possibili sviluppi verranno presentati nell'ultimo capitolo.

Capitolo 2

Le reti a pacchetto

Attualmente tutte le comuni reti di calcolatori sono basate sulla commutazione di pacchetto, un nuovo approccio alla comunicazione fra entità differenti introdotto negli anni '60 e che oggi permette di interfacciare fra loro calcolatori posizionati in vari punti del globo attraverso Internet. Il concetto fondamentale alla base di questa infrastruttura è appunto quello di *pacchetto*, inteso come frammento di informazione digitale che viene instradato secondo numerosi criteri lungo la rete di calcolatori.

Questo approccio offre numerosi vantaggi rispetto alla commutazione di circuito, precedentemente impiegata nelle reti telefoniche, che risulta meno efficiente ed economica.

2.1 Origini

L'origine dello sviluppo delle reti a commutazione di pacchetto è partita grazie al lavoro di Paul Baran e Donald Davies, due ricercatori che, in maniera indipendente, introdussero e svilupparono il concetto di *packet switching*, precedentemente anticipato dalle pubblicazioni di Leonard Kleinrock nel campo della commutazione digitale.

Baran si occupò di definire i principi alla base delle reti presenti alla RAND Corporation, in parte finanziata dalla U.S. Air Force, nell'ambito dell'indagine sull'impiego delle comunicazioni numeriche in ambito militare [1]. Baran delineò una generica architettura progettata per essere largamente scalabile, altamente affidabile e in grado di continuare ad operare il caso di guasti o malfunzionamenti. Il tutto può essere riassunto in alcuni punti fondamentali, fra i quali adottare una rete decentralizzata e possibilmente ridondante, ovvero con più di un cammino fra un nodo e l'altro.

Nello stesso periodo, Donald Davis, un ricercatore inglese presso il Laboratorio Nazionale di Fisica, concepì autonomamente il concetto di rete a commutazione di pacchetto, che pensava di impiegare per la costruzione di un network globale oltremanica. In seguito, i due gruppi di ricerca si unirono, ma è curioso osservare come fossero arrivati alle stesse conclusioni con parametri confrontabili, nonostante non avessero mai scambiato i risultati.

2.2 Funzionamento

Dal punto di vista pratico, una generica rete di comunicazione è formata da due elementi fondamentali: i canali di comunicazione (*link*) e i nodi. Possiamo quindi considerare la rete come un sistema complesso di porzioni via via più semplici, sino ad arrivare al più elementare possibile, ovvero il canale di trasmissione. I nodi intermedi hanno quindi la funzione di inoltrare i pacchetti che transitano sulla rete attraverso i link presenti, al fine di recapitare le informazioni dall'end-point di partenza a quello di arrivo.

Più precisamente, la commutazione di pacchetto può essere vista come una tecnica di multiplexing statico nella quale il singolo canale di trasferimento viene suddiviso in una serie di sotto-canali logici, ognuno dedicato ad un flusso (*stream*) di dati. Il messaggio oggetto della comunicazione viene suddiviso in frammenti (denominati appunto pacchetti), ognuno dei quali reca una serie di informazioni aggiuntive, utili al corretto invio attraverso la rete, che in questo modo può avvenire in maniera del tutto indipendente dagli altri. La differenza principale rispetto al precedente tipo di commutazione sta proprio nel ruolo attivo dei nodi intermedi, che in base ai dati contenuti nell'header, decide se e in che modo inoltrare il pacchetto.

La tecnica di instradamento lungo i percorsi disponibili viene determinato in base ad uno specifico algoritmo, detto di *routing*, che può essere progettato per utilizzare diverse strategie e figure di merito.

2.3 Architetture a strati

Nel corso dei primi sviluppi delle reti a pacchetto ci si è accorti che la loro realizzazione tramite un approccio di tipo monolitico introduceva una serie di problemi in grado di inficiare il funzionamento e il mantenimento di tali infrastrutture. Con questa strategia sarebbe necessario installare sui noti intermedi del software ad-hoc per ogni trasferimento e rimpiazzarlo ogni qual volta cambiano le applicazioni sugli endpoint o la topologia della rete, cosa chiaramente irrealizzabile.

Per migliorare la flessibilità del sistema si è quindi optato per una soluzione a livelli, ognuno dei quali si occupa dei vari aspetti del processo di trasmissione e realizza funzioni più complesse di quello precedente. Ogni layer si appoggia a quello sottostante per usufruire dei suoi servizi, che vengono poi integrati con delle funzionalità aggiuntive più complesse. Si crea in questo modo uno *stack*, denominato appunto stack di rete, al quale si appoggiano le varie applicazioni che hanno necessità di eseguire un trasferimento di informazioni.

Col passare del tempo sono stati proposti numerosi modelli per implementare questo tipo di struttura, sino a giungere al modello di riferimento tuttora utilizzato, il modello ISO/OSI [2], che definisce sette layer che rappresentano, dal livello di astrazione più alto (applicativo) a quello più basso (fisico), la trasmissione dei dati. Gli strati intermedi si occupano di garantire servizi essenziali per la comunicazione, quali per esempio l'integrità e il corretto ordinamento dei pacchetti, l'indirizzamento fisico e logico, e funzionalità aggiuntive come la criptazione dei dati e la loro rappresentazione.

Un principio cardine di questo approccio è l'indipendenza fra i vari livelli.

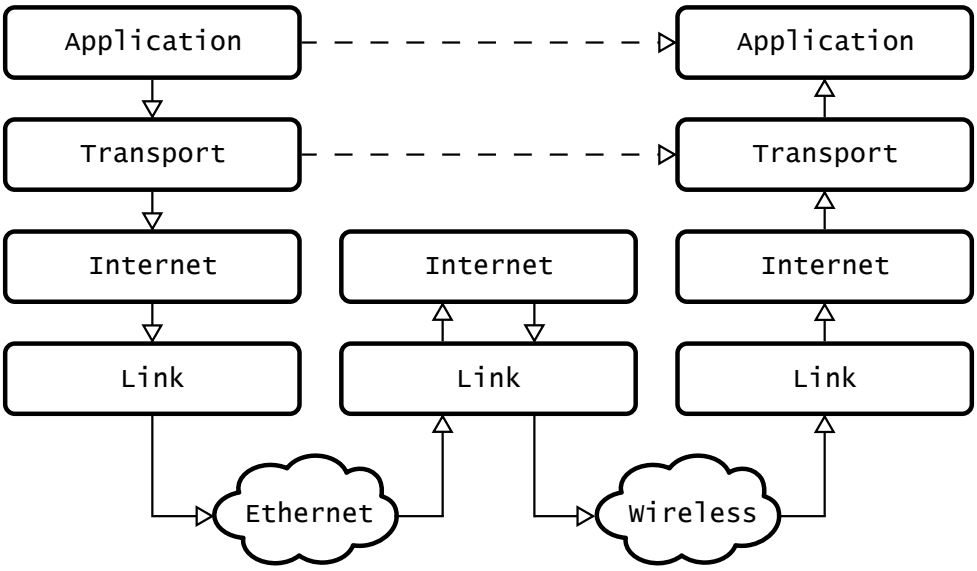

Figura 2.1: Inoltro dei pacchetti all'interno dello stack TCP/IP

Ogni livello deve essere in grado di lavorare senza far riferimento a informazioni caratteristiche degli altri livelli (cosa che in realtà non è sempre rispettata nelle implementazioni reali), il che permette per esempio di sostituire uno strato con uno equivalente, purché sia mantenuta la stessa interfaccia fra un layer e l'altro[1].

2.3.1 Stack TCP/IP

Il modello ISO/OSI appena esposto fornisce uno scheletro concettuale, attorno al quale sono state realizzate numerose implementazioni nel corso degli anni, quali SNA di IBM [3] e DNA di Digital [4]. L'architettura ad oggi di gran lunga più utilizzata è quella TCP/IP, che prende il nome dai due protocolli fondamentali, impiegati rispettivamente a livello 4 e 3. La suite TCP/IP ha registrato fin da subito un enorme successo, dovuto principalmente alla sua essenzialità ed efficienza, ed al fatto che sia una tecnologia non proprietaria e sia stata di conseguenza adottata in progetti molto importanti come UNIX ed ARPANet.

Lo stack TCP/IP non ricalca in maniera fedele quanto è stato precedentemente esposto riguardo all'ISO/OSI, anzi sono presenti alcune differenze fondamentali; innanzitutto la differenziazione e l'indipendenza fra i livelli non è strettamente rispettata, in quanto considerata *dannosa* [5]. Inoltre lo stack TCP/IP consiste in solamente quattro livelli: lo strato *Application*, che riunisce grossomodo i tre livelli superiori ISO/OSI, lo strato *Transport*, lo strato *Internet* ed infine lo strato *Link*, che prende il posto dei due strati più bassi nello stack ISO/OSI (Figura 2.1).

Attualmente questa architettura rappresenta lo standard *de facto* nell'industria delle comunicazioni informatiche, in quanto è adottata per interconnettere i nodi che costituiscono la rete Internet e la maggior parte delle reti locali (LAN); è quindi di fondamentale importanza la sua comprensione per la trattazione e lo sviluppo degli argomenti che seguiranno.

[1] Concetto mutuato dalla programmazione, che afferma che un componente è caratterizzato dalla sua interfaccia e non dalla sua implementazione

2.4 Routing

In generale, dati due nodi di rete fra i quali è instaurato uno scambio di pacchetti, sono possibili numerosi cammini per la trasmissione dei dati. L'atto di definire (in maniera statica o dinamica) un percorso (*route*) da seguire viene denominato *routing* o instradamento. L'operazione non è associata ad alcun strato protocollare specifico, ma è solitamente eseguita a livello 3 (*Network*) e talvolta a livello 2 (*Data Link Control*).

Esistono numerosi algoritmi di routing, che si dividono principalmente in due categorie: statici e dinamici.

2.4.1 Algoritmi statici

Questa famiglia di algoritmi è caratterizzata dal fatto che i risultati forniti non cambiano al variare delle caratteristiche della rete (topologia, traffico, ecc...). Il criterio usato per la scelta dei cammini può essere di qualsiasi tipo e viene definito al momento della configurazione della rete, durante la quale l'amministratore di rete riempie manualmente particolari tabelle, dette appunto di routing, nelle quali vengono descritti tutti i particolari percorsi e le relative destinazioni. Nonostante questo approccio sia molto limitato e si possa applicare solo a topologie di rete molto semplici non soggette a modifiche, offre indubbiamente alcuni vantaggi, essendo molto semplice da realizzare e introducendo un traffico di rete aggiuntivo (*overhead*) trascurabile.

2.4.2 Algoritmi dinamici

In opposizione agli algoritmi statici appena descritti, esiste una tipologia che invece varia i propri risultati in base allo stato in cui si trova la rete. Essi vengono definiti algoritmi dinamici, progettati per fornire cammini validi in risposta al cambiamento di alcune condizioni o al verificarsi di alcuni eventi, come per esempio il danneggiamento di un nodo di rete, che causerebbe l'invalidazione di tutti percorsi attraverso di esso in caso di routing statico. Per questo motivo, questi algoritmi sono particolarmente indicati per l'utilizzo in reti di grosse dimensioni o soggette a modifiche frequenti. I principali svantaggi sono dovuti all'elevata complessità del procedimento di routing, implementato solitamente in maniera automatica attraverso protocolli appositamente progettati (ad esempio RIP [6], OSPF [7] e IS-IS [8]), che introducono un traffico di rete aggiuntivo.

2.5 Vantaggi

Come già detto in precedenza, l'impiego delle reti a pacchetto offre la possibilità di ottimizzare numerosi aspetti della comunicazione, come l'utilizzo dei singoli canali, la riduzione delle latenze e l'aumento della robustezza dell'infrastruttura.

2.5.1 Efficienza

Rispetto alla vecchia commutazione di circuito, nella quale l'intero canale di comunicazione viene impiegato per la trasmissione di un solo flusso di informazioni[2], la commutazione a pacchetto permette di effettuare dividere in maniera ottimale la banda fra i vari utilizzatori, affinché ognuno possa percepire, nel caso ideale, l'intera banda disponibile. È quindi possibile ridurre il numero di canali di trasmissione utilizzati, con conseguente riduzione dei costi e aumento della semplicità (e della mantenibilità) della rete.

2.5.2 Performance

La suddivisione in pacchetti del traffico da inviare offre un ulteriore beneficio, derivante da un fenomeno noto come *pipelining*: i vari nodi intermedi e i relativi canali sono in grado di effettuare la trasmissione in contemporanea e, analogamente ad una catena di montaggio, il pacchetto in output da un nodo diventerà poi l'input del nodo successivo presente lungo il percorso. In questo modo si ottiene una maggiore efficienza, dovuta ad un utilizzo migliore della capacità della rete, nonché una diminuzione notevole del ritardo di trasmissione rispetto ad una rete che non divide i messaggi in pacchetti.

2.5.3 Affidabilità

La robustezza è sicuramente uno dei vantaggi principali delle reti a pacchetto. Come già visto nella Sezione 2.4, solitamente esistono numerosi cammini per congiungere due end-point specifici. Questa *ridondanza* è garanzia di affidabilità, perché anche in caso di malfunzionamento o caduta di un nodo intermedio i pacchetti possono attraversare ugualmente la rete lungo uno dei vari percorsi alternativi. Inoltre, la rete può essere configurata per reagire in maniera dinamica ai mutamenti della propria topologia attraverso l'uso di algoritmi dinamici.

Questo fu il motivo che spinse il progetto ARPANet ad adottare la commutazione di pacchetto: all'epoca, infatti, i nodi intermedi erano poco affidabili ma era comunque necessario fornire ai vari ricercatori una buona connettività fra i vari super calcolatori allora disponibili.

Ulteriore garanzia di sicurezza è data dai vari meccanismi di rilevamento e correzione d'errore, adottati da particolari protocolli al fine di assicurare l'integrità dei dati trasmessi. Questo fatto è particolarmente importante nel contesto di questa trattazione, dato che per la creazione e l'affinamento delle tecniche per prevedere la perdita dei pacchetti sono necessari strumenti che generano tale problema in maniera controllata. Questa tematica verrà trattata dettagliatamente nel successivo Capitolo 3.

[2]Per n flussi contemporanei servono n canali

Capitolo 3

Il packet loss

Le reti a pacchetto, in quanto mezzi di telecomunicazione, non sono immuni da tutte le problematiche che affliggono questo ambito: rumore sul canale trasmissivo, ritardo nella ricezione, perdita o aggiunta di informazioni, corruzione del segnale, sono tutti aspetti da cui è impossibile prescindere e che vanno affrontati in maniera corretta.

In questo capitolo ci occuperemo di un fenomeno frequente che affligge le reti di calcolatori, il *packet loss* o perdita di pacchetti, mostrandone le cause, gli effetti, come riprodurlo e soprattutto come evitarlo.

3.1 Cause

La perdita di pacchetti in una rete di calcolatori può avere numerose cause di natura differente. Il segnale può degradarsi per effetto di un canale di trasmissione rumoroso, come un cavo danneggiato o una rete wireless soggetta ad interferenze, oppure non arrivare con una potenza sufficientemente elevata.

Possono inoltre sorgere problemi sui nodi intermedi, che possono congestionarsi e quindi scartare i pacchetti una volta che il buffer di ricezione è pieno, o semplicemente operare in maniera scorretta a causa di un hardware difettoso.

Inoltre, anche se il comparto fisico funziona correttamente, la perdita di pacchetti può essere causata da errori software, come algoritmi di routing mal codificati (in tal caso i pacchetti vanno semplicemente perduti) oppure device driver realizzati in maniera errata.

3.2 Effetti

Quando causata da problemi strutturali, la perdita di pacchetti può portare ad un degradamento significativo delle prestazioni della rete, con una visibile riduzione della velocità di trasferimento, un aumento del delay di ricezione, e l'eventuale comparsa del fenomeno noto come *jitter*[1] (come si vede in Figura 3.1), cosa molto spiacevole per applicazioni multimediali, quali VoIP, videoconferenza, streaming video e gaming online. Va comunque precisato che il packet loss, in piccola misura,

[1] Elevata variabilità del tempo necessario ad un pacchetto per attraversare la rete

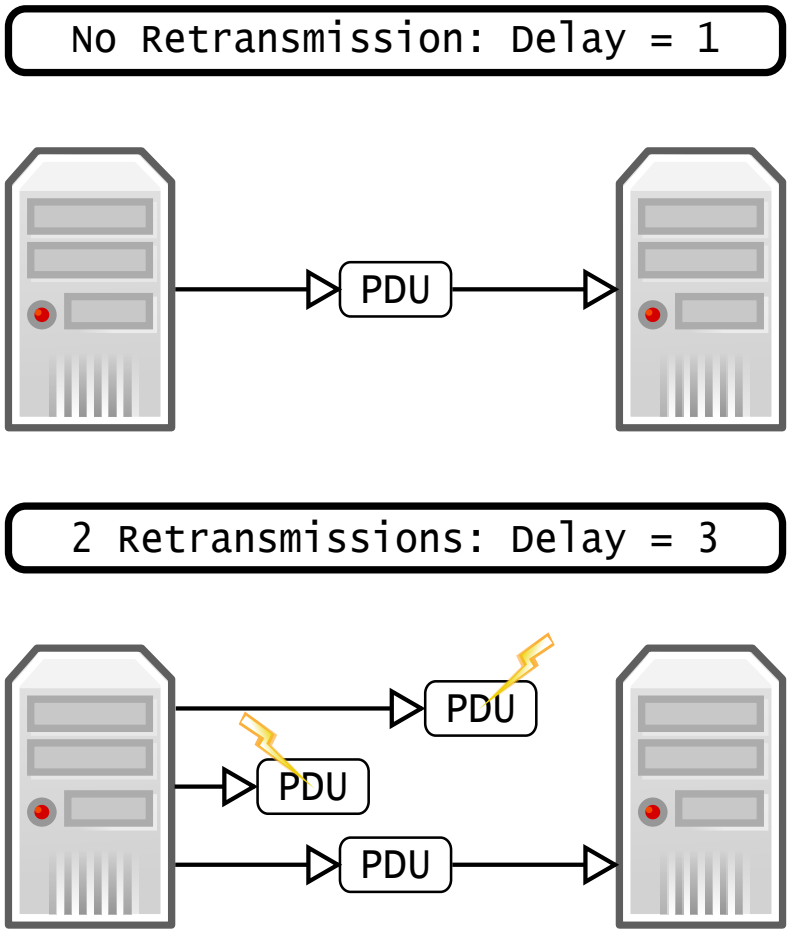

Figura 3.1: Correlazione fra pacchetti perduti e jitter

può essere tollerato e non porta a conseguenze tangibili, soprattutto se si verifica a livello dei nodi intermedi ma non si riscontra poi nell'end-point.

3.3 Soluzioni

La rilevazione e la correzione degli errori di trasmissione svolgono quindi un ruolo fondamentale nelle applicazioni di rete. Esistono sostanzialmente due filosofie per affrontare il problema della perdita dei pacchetti: scartare il pacchetto errato e richiedere la ritrasmissione dello stesso, oppure cercare di ricostruire in maniera soddisfacente le informazioni originali dai dati (incompleti o errati) che sono giunti a destinazione. Nelle prossime sezioni vedremo come questi due approcci ambivalenti si traducono in specifiche implementazioni lungo lo stack TCP/IP, in particolare a livello applicativo e *Transport*.

3.3.1 TCP

Il protocollo TCP (*Transmission Control Protocol*) [9] è in grado di garantire una trasmissione affidabile di pacchetti attraverso determinati controlli, eseguiti fra l'host di partenza e quello di destinazione (*end-to-end*). Il TCP è un protocollo

orientato alla connessione, in quanto ogni scambio di dati fra due entità comunicanti è inserito in un flusso logico di informazioni, chiamato appunto connessione, che viene poi esplicitamente chiusa quando non è più necessaria.

La correttezza dei dati inviati viene assicurata tramite un sistema di conferma dell'avvenuta ricezione, implementata tramite appositi pacchetti detti *ACK*. La politica di gestione di queste conferme è importante perché si rischia di inondare la rete con dati non necessari, soffocandone le prestazioni. A tal proposito, il TCP adotta un sistema di ACK cumulativi, usati per validare tutti i pacchetti inviati sino a quel momento. In casi particolari è inoltre disponibile un sistema di ACK selettivi, riferiti cioè a sequenze discontinue di pacchetti, il che evita inutili ritrasmissioni di dati già ricevuti correttamente.

Ordine di ricezione

Il TCP adotta uno schema di numerazione per identificare i byte inviati da un host ad un altro, accorgimento che permette, sul nodo di destinazione, di ricostruire i dati in maniera corretta, a prescindere da eventuali frammentazioni, riordinamenti o perdite di pacchetti. Ogni qual volta viene inviato un nuovo payload, il numero di sequenza viene incrementato, a partire da un valore pseudo-casuale scelto durante la fase di instaurazione della connessione (*3-way handshake*), eseguita all'inizio del trasferimento per sincronizzare lo stato fra i due host. È importante che il valore iniziale non sia prevedibile, per evitare i cosiddetti attacchi *man-in-the-middle*, realizzati in questo caso tramite la contraffazione del numero di sequenza al fine di iniettare dati estranei nella connessione oppure chiudere prematuramente la stessa tramite l'invio di un pacchetto adeguatamente costruito.

Il TCP offre comunque alcune protezioni contro questo genere di attacchi, che includono il controllo della coerenza degli orari di trasmissione e delle informazioni provenienti dai livelli protocollari inferiori. Se i dati non coincidono, il pacchetto viene marcato come corrotto e viene scartato.

Rilevazione degli errori

Per verificare la correttezza delle PDU ricevute, il TCP riserva un campo nella testata dei pacchetti, chiamato *checksum*, nel quale viene memorizzata una somma di controllo calcolata sui dati di partenza. Una volta ricevuti dal destinatario, essi vengono nuovamente processati con lo stesso algoritmo per assicurarsi che i due risultati coincidano, altrimenti il pacchetto risulta scorretto e viene rifiutato. La procedura per il calcolo del checksum è volutamente semplice, scelta giustificata da alcune considerazioni: utilizzare un algoritmo più complesso richiederebbe del lavoro aggiuntivo per la CPU, che verrebbe occupata inutilmente, dato che un controllo più robusto è implementato a livello 2. Questo però non rende il checksum TCP inutile, dato che possono generarsi comunque errori anche in trasmissioni che si appoggiano su strati DLC con controllo di integrità ciclico (*CRC*). Infatti, se esso risulta quasi infallibile in presenza di rumore non deterministico, si osservano comunque pacchetti che falliscono il controllo di integrità con una incidenza molto elevata [10]. Le cause si inseriscono lungo tutto lo stack TCP/IP, e includono errori di memoria e bug.

Controllo del flusso

Come è stato detto in precedenza, può essere necessario regolare la velocità di invio dei dati verso un host incapace di processarli ad un ritmo troppo elevato, al fine di evitare il riempimento del buffer di ricezione e la conseguente perdita di pacchetti.

Il TCP adotta un meccanismo di controllo di flusso chiamato "finestra scorrevole", che prevede la definizione di una finestra di ricezione, con la quale il destinatario specifica il quantitativo di byte che è disposto ad accettare ulteriormente. Se il numero (memorizzato nella testata dei pacchetti) è diverso da zero significa che l'host di partenza è autorizzato ad inviare quella determinata dimensione di dati, in caso contrario esso deve bloccarsi fino alla ricezione di nuovi ACK che confermano le PDU precedenti (Figura 3.2). Quando questo accade viene attivato un meccanismo per scongiurare il verificarsi di una situazione di *deadlock*[2], che potrebbe avvenire se il messaggio di incremento della finestra di ricezione va perduto. Sul nodo d'invio viene azionato un timer, al termine del quale viene inviato un piccolo pacchetto la cui conferma contiene la nuova dimensione della finestra.

Unitamente a questo, il TCP mette in atto un controllo per evitare il collasso non solo del ricevente, ma anche dei nodi intermedi che svolgono l'inoltro dei pacchetti. È possibile infatti dedurre lo stato di congestione della rete dalla frequenza (o dall'assenza) delle conferme che giungono al nodo di partenza. Queste informazioni, combinate con l'uso di appositi timer inseriti nelle testate dei pacchetti, permettono al TCP di regolare il ritmo di invio dei dati e di evitare il cosiddetto "collasso da congestione", attraverso quattro algoritmi appositamente progettati per lo scopo [11].

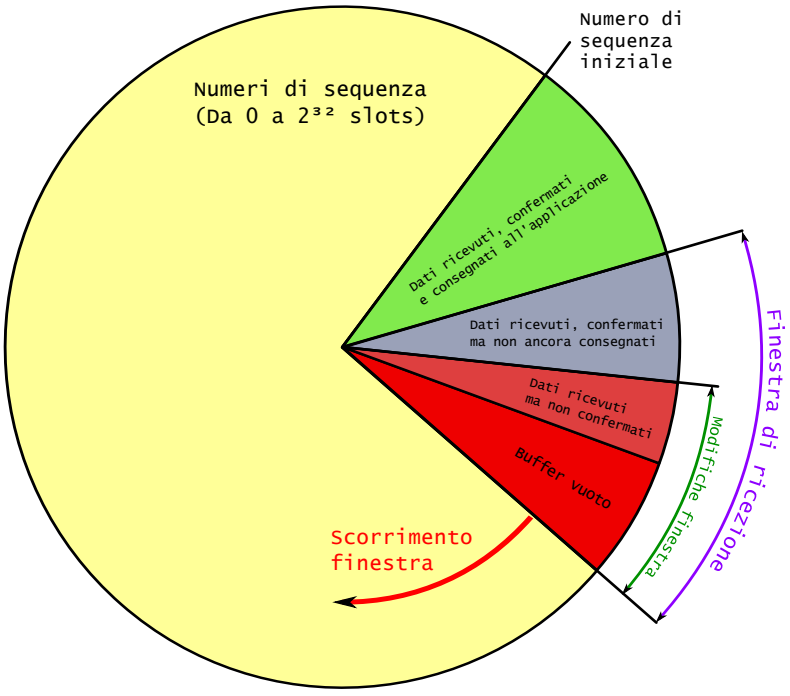

Figura 3.2: Variazione della finestra di ricezione

[2]Situazione di stallo in cui entrambi i partecipanti aspettano che l'altro esegua un'azione per proseguire

3.3.2 UDP

In molte applicazioni, tuttavia, l'uso del protocollo TCP non è appropriato. Non è possibile, per esempio, ottenere i pacchetti ricevuti correttamente dopo un errore di invio finché quest'ultimo non viene risolto tramite la ritrasmissione dello stesso. Questo può essere spiacevole per applicazioni *real-time*, come lo streaming multimediale via Internet, giochi multiplayer online e il VoIP (Voice Over IP), dove è più importante ottenere basse latenze piuttosto che ricevere tutti i dati nell'ordine corretto. La complessità del TCP può essere un problema in applicazioni integrate (*embedded*) o per server che devono gestire un numero elevatissimo di richieste semplici, come nel caso dei server DNS.

In questi casi l'uso del TCP viene rimpiazzato dal protocollo UDP (*User Datagram Protocol*) [12], che fornisce le stesse funzionalità di *multiplexing*[3] applicativo e rilevazione degli errori, ma non si occupa della ritrasmissione e del riordino del pacchetti. Questo permette di ottenere una gestione più snella e veloce dei trasferimenti, caratteristiche chiave nelle applicazioni citate precedentemente. Inoltre l'UDP non instaura connessioni all'inizio di ogni sessione di trasferimento, al contrario ogni messaggio (*datagram*) viene inviato come informazione a sé stante, senza conservare dettagli sullo stato dei due nodi interlocutori.

L'UDP adotta quindi una politica di tolleranza, limitandosi a controllare, tramite l'uso di un *checksum*, la coerenza dei dati. La loro eventuale correzione o ritrasmissione viene invece completamente delegata a livello applicativo, dove si possono adottare tecniche di interpolazione oppure codici a correzione d'errore, come vedremo nella sezione successiva.

Dato che l'UDP ignora anche le problematiche riguardanti il controllo del flusso e della congestione, essi devono essere implementati manualmente in altra forma, con tecniche specifiche che variano da situazione a situazione. Recentemente è stato sviluppato un apposito protocollo, denominato DCCP (*Datagram Congestion Control Protocol*) [13], che sgrava l'applicazione dal farsi carico delle politiche di gestione del flusso dei dati.

3.3.3 Livello applicativo

Come abbiamo visto, l'impiego di un protocollo a livello *Transport* di tipo confermato come il TCP, porta ad evitare la perdita di pacchetti tramite la ritrasmissione, cosa che non sempre è possibile per esigenze di latenza e velocità in campi come per esempio il VoIP e lo streaming online. In questi ambiti si implementano di solito controlli a livello applicativo per usufruire della reattività del protocollo UDP e mantenere al tempo stesso una buona qualità nel segnale ricevuto.

È possibile per esempio ricorrere ai cosiddetti codici a correzione d'errore, che tramite particolari algoritmi matematici, possono ricostruire correttamente i dati danneggiati, mediante l'uso di informazioni aggiuntive allegate ai dati, dette ridondanze [14]. Questa tecnica è di fondamentale importanza in ambiti in cui la ritrasmissione è di fatto impraticabile (come nella memorizzazione di dati su supporto fisico), ma non trova molta applicazione nel campo delle reti di calcolatori, dato che presenta una serie di costi aggiuntivi elevati in rapporto ai benefici offerti.

[3] Capacità di gestire richieste da più applicazioni contemporaneamente

In campo multimediale si è soliti ricorrere a tecniche particolari, atte a ricostruire i dati danneggiati sulla base delle informazioni ricevute. In ambito video è possibile, per esempio, derivare i frame mancanti mediante interpolazione fra l'immagine precedente e quella successiva; lo stesso principio viene applicato, in maniera differente, in campo audio: analizzando la forma d'onda e ricercando i modelli ricorrenti nel segnale vocale, è possibile colmare il gap di informazioni mancanti senza un'apparente degradazione della qualità per l'utente finale.

3.4 Realizzazione di lossy link

Vista l'esigenza di disporre di meccanismi per contrastare la perdita dei pacchetti nelle reti di calcolatori, è altrettanto importante per gli sviluppatori avere a disposizione degli strumenti che permettano di simulare, in maniera controllata, il problema in oggetto, al fine di eseguire test e ottimizzare gli algoritmi fin qui esposti.

Prima di esporre l'approccio attraverso le interfacce virtuali oggetto di questa tesi, descritte nel Capitolo 4, vediamo alcune alternative messe a disposizione dal sistema operativo Linux.

3.4.1 Driver ad-hoc

Una possibile soluzione consiste nell'integrare completamente i meccanismi che ricreano la perdita di pacchetti a livello device driver, modificando in maniera appropriata la parte di trasmissione, in modo che inoltri solamente una porzione dei dati da inviare secondo il modello selezionato dall'utente.

Questo approccio, nonostante sia molto efficiente grazie al livello di astrazione molto basso, soffre di alcuni difetti che lo rendono, di fatto, difficile da implementare. Esso è infatti legato strettamente al tipo di hardware utilizzato e quindi presenta un grado di portabilità bassissimo: una volta sostituita l'interfaccia di rete (o anche semplicemente il driver ad essa associato), è necessario modificare nuovamente il software sulla base della nuova architettura hardware. Questo sforzo è quindi giustificabile solamente nel caso di dispositivi estremamente semplici o con elevate richieste in termini di prestazioni, come applicazioni integrate o real-time, per le quali può avere senso sviluppare un driver *ad-hoc*.

Per motivi di test, questa alternativa è stata comunque approfondita, applicandola ad un device driver Fast Ethernet Atheros L1E, il cui funzionamento è stato mutuato direttamente da Insane. La perdita di pacchetti viene di fatto realizzata all'interno della funzione `atl1e_xmit_frame()` (Appendice A.4), nella quale è stata inserita la logica che determina se un pacchetto deve essere inoltrato o no: in caso affermativo si procede con le normali routine di trasmissione del driver, altrimenti lo si scarta.

Per la configurazione dei parametri del modello di perdita è stata utilizzata la struttura adottata in tutti i tool esposti in questa sezione, che prevede un componente posizionato nello spazio kernel (il driver, appunto), al quale vengono inoltrati i parametri di funzionamento da un programma utente, che in questo caso agisce tramite la chiamata di sistema `ioctl` (Sezione 4.2.4). Questo meccanismo

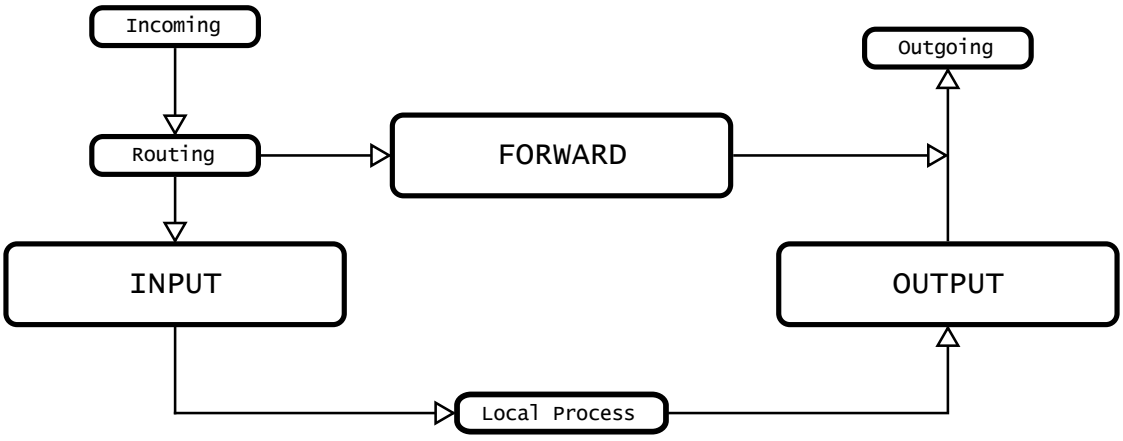

Figura 3.3: Posizione delle tre catene fondamentali di Iptables

permette di configurare in maniera rapida il driver, il tutto senza la necessità di modificare e ricompilare il codice sorgente.

Nonostante questo approccio sia teoricamente realizzabile e di fatto funzionante, non è stato poi comparato alle alternative qui illustrate a causa della sua bassa portabilità, che ha reso molto difficoltosa la sua implementazione nelle configurazioni usate per testare le performance di Insane, descritte in dettaglio nel Capitolo 5.

3.4.2 Iptables

Per ovviare al problema di portabilità dell'approccio device driver si può ricorrere alle funzionalità fornite dal sistema operativo per ricreare la perdita dei pacchetti in maniera controllata. Una possibilità consiste nel far ricorso a Iptables/Netfilter [15], un framework presente a livello kernel che permette all'amministratore di controllare, attraverso una serie di regole riunite in tabelle, il percorso seguito dai pacchetti durante il loro attraversamento dello stack di rete.

Iptables fa parte di un'infrastruttura che abbraccia più strati protocollari, denominata *Xtables* (da `x_tables`, modulo kernel che riunisce il codice condiviso da tutti componenti), che oltre alle funzionalità IPv4, permette di agire anche sui protocolli IPv6, ARP ed Ethernet.

Il funzionamento è basato su una serie di checkpoint, detti *hook*, presenti a più livelli nello stack, che vengono via via richiamati al passaggio di un pacchetto, reindirizzato poi nella maniera specificata dall'amministratore di rete. Esistono numerosi hook predefiniti: i più importanti sono denominati `INPUT`, riferito ai pacchetti che sono destinati all'host locale, `FORWARD`, per i pacchetti che devono invece essere inoltrati tramite routing, ed infine `OUTPUT`, dal quale passano i dati generati localmente e poi inviati (Figura 3.3). Ad ognuno di essi è possibile associare una *catena*, intesa come serie di azioni chiamate *regole* che specificano, attraverso alcuni filtri, a quale tipologia di pacchetti si riferiscono e le azioni da compiere su di essi. Iptables è estremamente flessibile da questo punto di vista, permettendo di specificare minuziosamente le specifiche di filtraggio da applicare, quali l'indirizzo di partenza e di destinazione, il protocollo impiegato o l'interfaccia utilizzata.

Una volta che il pacchetto soddisfa i requisiti di una regola, viene attivato il

comportamento specificato dall'utente in fase di configurazione, che può comprendere il passaggio ad un'altra catena personalizzata, o l'uso di un modulo che estende le funzionalità di Iptables. Dato che Iptables nasce come strumento per regolamentare il flusso di pacchetti attraverso un nodo di rete, è consuetudine configurarlo da firewall, in modo che scarti tutti i pacchetti che sono considerati dannosi per il sistema, il che è reso possibile utilizzando il *target*[4] predefinito `DROP`. Per esempio, con un'istruzione del tipo:

```
iptables -A INPUT -s 192.168.0.1 -j DROP
```

si eliminano tutti i pacchetti che provengono dall'indirizzo 192.168.0.1, che è in questo caso considerato fonte di traffico pericoloso per il sistema.

Questo comportamento, opportunamente modificato, permette di realizzare la perdita di pacchetti controllata oggetto di questa trattazione. La componente casuale del fenomeno è ricreata tramite un'espansione di Iptables, chiamata *statistic*, che permette di definire regole che si applicano in maniera probabilistica con una data percentuale. Con il comando:

```
iptables -A FORWARD -d 192.168.2.1 -m statistic \
--mode random --probability 0.5 -j DROP
```

si realizza quanto esposto. Posta su un nodo che funge da router, questa regola si aggiunge alla catena `FORWARD` (dato che si applica ai pacchetti da inoltrare) scartando, con una probabilità del 50%, il traffico verso l'indirizzo 192.168.2.1.

3.4.3 Netem

Il framework Netem (*Network Emulator*) [16] nasce originariamente con il semplice intento di testare e ottimizzare il comportamento del TCP nei confronti del ritardo dei pacchetti ed ha via via integrato numerose funzionalità aggiuntive, fino a diventare un tool sofisticato per l'emulazione di reti di grossa dimensione (*WAN*).

Il suo funzionamento è basato su principi diversi rispetto ad Iptables. Netem agisce infatti come uno *scheduler* di rete, ovvero determina le politiche con le quali le code di pacchetti in attesa di essere inviati vengono gestite. Queste specifiche, chiamate `qdisc`, costituiscono i mattoni fondamentali con le quali il traffico di rete viene controllato dal sistema operativo.

Il kernel Linux fornisce numerose politiche di schedulazione, consultabili nella cartella `net/sched` del codice sorgente. Esse possono variare dalla più semplice, di tipo FIFO, a modelli più complessi, nei quali si applicano concetti più avanzati di gestione delle code. La versione predefinita usata dal sistema operativo prevede la suddivisione dei pacchetti in classi di priorità, che vengono gestite internamente con logica lineare, fra le quali viene instaurata una rigida gerarchia; le classi meno prioritarie non vengono soddisfatte finché quelle superiori non sono state esaurite (Figura 3.4). Esistono inoltre algoritmi che introducono meccanismi di rotazione all'interno della coda, come lo *Stochastic Fairness Queuing*, che è impiegato per fornire, in maniera imparziale, la banda disponibile ad una serie arbitraria di flussi di informazioni.

Netem, mediante l'uso di questi automatismi, permette ricreare con precisione i parametri propri delle reti di notevole estensione, caratterizzate da alti valori di

[4]Destinazione del pacchetto che soddisfa la regola

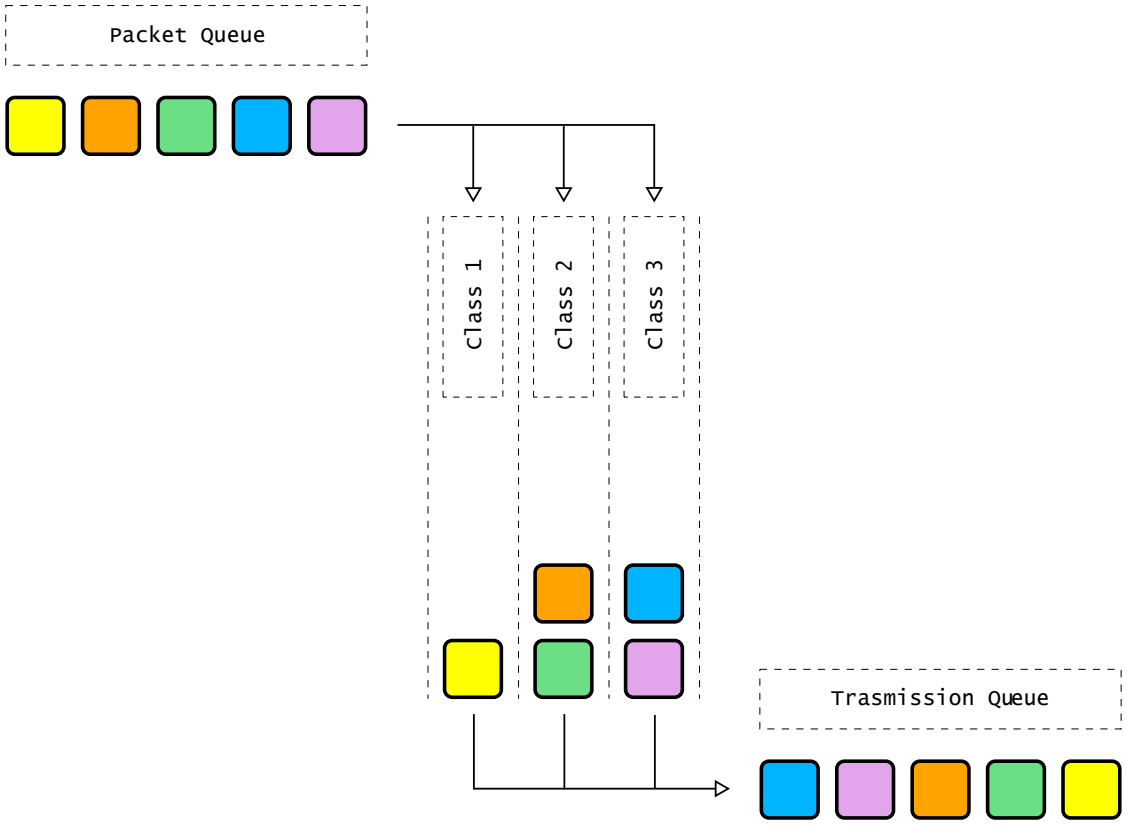

Figura 3.4: Politica predefinita per la gestione delle code su Linux

ritardo nei pacchetti, oltre che perdita, duplicazione e riordino degli stessi. Esso, analogamente ad Iptables, è implementato attraverso un modulo kernel, chiamato `sch_netem`, che viene poi configurato a runtime tramite il tool a riga di comando `tc`, facente parte del pacchetto *iproute2*. Questo programma permette di creare nuove politiche di schedulazione da associare a specifiche interfacce di rete, nelle quali è possibile introdurre una componente statistica per aumentare la precisione della simulazione. Per esempio, con il comando:

```
tc qdisc add dev eth0 root netem delay 100ms 10ms 25%
```

si applica un ritardo ad ogni pacchetto variabile da 90 ms a 110 ms, nel quale ogni valore è soggetto ad una correlazione del 25% con il precedente.

È inoltre possibile realizzare, tramite la direttiva `loss`, una perdita casuale di pacchetti, che viene specificata tramite percentuale ed un eventuale correlazione. In tal caso, ogni generazione casuale di probabilità dipenderà nella frazione specificata dal valore precedentemente ottenuto. Ai fini della trattazione, tuttavia, ci limiteremo ad esaminare il caso più semplice, usato come riferimento nei test comparativi; per scartare la metà dei pacchetti in transito è necessario usare il seguente comando:

```
tc qdisc add dev eth0 root netem loss 50%
```

Capitolo 4

Insane Device Driver

PASSIAMO ora ad esporre in dettaglio le fasi di sviluppo e le caratteristiche del device driver Insane oggetto di questa tesi. Dopo una breve sezione introduttiva, atta ad esporre alcuni concetti di base ripresi poi in seguito, passeremo a dettagliare la struttura e l'impostazione del software, specificandone le modalità di sviluppo, di funzionamento e di configurazione.

4.1 Linux Device Driver

Per una comprensione approfondita degli argomenti che seguiranno è di fondamentale importanza capire cosa significa sviluppare un device driver operante su Linux. Come è lecito aspettarsi, esso dovrà essere pensato e costruito sulla base dell'architettura software sulla quale si appoggia e, quindi, non può prescindere da un'approfondita conoscenza della struttura interna del nucleo operativo del sistema.

Per questo motivo, la realizzazione di un generico device driver si traduce sostanzialmente nell'implementazione di un'*interfaccia* fornita dal kernel, che si limita a fornire i *prototipi*[1] delle funzioni che utilizzerà poi per dialogare con il modulo software. Il compito del programmatore è quindi quello di compilare il corpo delle funzioni ed associare ad ogni implementazione la corrispondente dichiarazione fornita negli header del kernel.

Le interfacce sono quasi sempre organizzate in strutture (*struct*), ovvero componenti software che racchiudono al loro interno dati di tipo eterogeneo, in questo caso riferimenti funzioni. Ogni driver può scegliere di implementare una o molteplici interfacce, a seconda delle funzionalità che intende offrire. A titolo di esempio, riportiamo un frammento di codice relativo a `file_operations`, adottato dai device che richiedono un accesso al file system:

```
struct file_operations {
        ...
        ssize_t (*read) (struct file *, ...);
        ssize_t (*write) (struct file *, ...);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        ...
};
```

[1] Dichiarazione che omette il corpo della funzione, riportando solamente parametri accettati e il tipo di ritorno

Come si può notare, i nomi scelti sono significativi dell'operazione che l'istruzione deve eseguire. Ogni volta che il kernel ha necessità di eseguire un operazione di lettura, per esempio, richiamerà la funzione `read` eseguendo il codice scritto nel relativo corpo. Per questo motivo, come introdotto in precedenza, è necessario specificare manualmente questa associazione; il compilatore `gcc` permette di adottare la sintassi definita dalla revisione C99 del linguaggio C [17], semplificando significativamente questa fase:

```
struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};
```

Tuttavia è lecito utilizzare la sintassi standard di assegnazione in maniera equivalente, come mostrato nella Sezione 4.2. Infine, per rendere il driver parte integrante del sistema operativo arricchendone le funzionalità, è necessario caricare il codice compilato in memoria nel cosiddetto spazio kernel. A tal proposito, Linux offre un meccanismo semplice ed efficace che fa ricorso ai cosiddetti *moduli kernel*, che verranno trattati dettagliatamente nella sezione successiva.

4.1.1 Moduli Kernel

Linux fa parte della famiglia dei kernel monolitici, caratterizzati dal fatto che l'intero sistema operativo è eseguito nel cosiddetto spazio kernel, definendo un'unica interfaccia di alto livello sopra lo strato hardware, e fornendo una libreria di chiamate di sistema per gestire servizi quali gestione dei processi, concorrenza, accesso alle periferiche e alla memoria.

È inoltre possibile ricorrere all'utilizzo dei cosiddetti *kernel modules* o moduli kernel, che permettono di caricare e rimuovere dinamicamente porzioni di codice precompilato all'interno del sistema operativo durante la fase runtime. È bene notare che questa modularità non è a livello di architettura (come avviene invece nei microkernel [18]) ma a livello binario: ciò non inficia la monoliticità di Linux, che può essere definito un kernel monolitico modulare.

Di fatto, questo meccanismo rappresenta un modo più intelligente e flessibile di gestire il nucleo del sistema operativo a runtime, senza dover ricompilare e ricaricare l'intero kernel.

Inoltre, l'uso di questo stratagemma permette di aumentare notevolmente l'efficienza in termini di occupazione di memoria. Un kernel interamente monolitico necessita di essere caricato nella sua interezza in RAM, anche se alcune porzioni di codice non sono immediatamente necessarie (driver di periferiche usate raramente, estensioni, ecc...). Un kernel modulare invece può ritardare l'allocazione dei moduli al momento in cui le loro funzionalità sono richieste, e procedere con la rimozione degli stessi una volta terminata l'esecuzione.

L'unico svantaggio è rappresentato dalla cosiddetta *frammentazione* della memoria; se caricato in blocco, infatti, il kernel è allocato in porzioni contigue di memoria dalle routine di installazione. Inserendo un nuovo modulo questa compattezza viene meno, dato che probabilmente il nuovo codice sarà caricato nel primo indirizzo disponibile, introducendo una piccola riduzione delle performance.

4.1.2 Kernel Space vs User Space

Come già accennato, una prerogativa dei kernel di tipo monolitico è la separazione della zona di memoria allocata dal kernel rispetto a quella destinata all'esecuzione dei normali processi utente. Questo approccio permette un migliore controllo di tutti gli aspetti cruciali del sistema, quali per esempio l'accesso all'hardware, la schedulazione dei task e la gestione della concorrenza. L'accesso diretto a queste funzionalità è infatti proibito ai singoli processi, che possono servirsi tuttavia di opportune chiamate fornite dal sistema operativo, come vedremo nella prossima sezione.

La "barriera" logica che separa il nucleo dai normali programmi è implementata mantenendo diversi spazi di indirizzamento all'interno della memoria centrale. Si definisce così il cosiddetto spazio kernel (*kernel space*), in opposizione allo spazio utente (*user space*). Capire questo aspetto è cruciale ai fini dello sviluppo di un device driver, che implica infatti un costante scambio di dati fra sistema operativo e tool user space, tematica che verrà trattata in dettaglio nella Sezione 4.3 dedicata ad Insanely.

Ogni qual volta è necessario copiare informazioni fra due spazi di indirizzamento diversi è quindi opportuno prestare estrema attenzione, dato che intervengono complesse logiche di traduzione fra indirizzi; fortunatamente il sistema operativo mette a disposizione specifiche funzioni che realizzano quanto detto, chiamate `copy_from_user()` e `copy_to_user()`.

Per ragioni di sicurezza, inoltre, non è ammesso ad un processo accedere ad un indirizzo esterno al proprio spazio utente, fatta eccezione per casi particolari quali *debugger*[2] o contesti nei quali una porzione di RAM condivisa fra programmi è stata esplicitamente definita. Esternamente alla propria area di memoria, un'applicazione non ha permesso d'accesso ed ogni tentativo genera la famosa eccezione a runtime denominata *segmentation fault*.

4.1.3 Chiamate di sistema

La suddivisione appena illustrata impone delle regole molto severe in termini di permessi delle singole applicazioni. Per esempio, l'accesso diretto all'hardware (inclusa memoria principale e secondaria) è proibito ai programmi, che quindi devono passare attraverso le funzionalità offerte dal sistema operativo per eseguire semplici operazioni, quali lettura e scrittura, per esempio.

L'interfaccia e tutte le funzioni fornite dal kernel ai singoli programmi utente vengono definite *system calls* o chiamate di sistema. Ogni programmatore utilizza (molte volte inconsapevolmente) moltissime system call, come è possibile verificare tramite appositi programmi che mostrano la lista delle chiamate effettuate, quali per esempio `strace`.

I moderni sistemi operativi dispongono inoltre di particolari librerie che si occupano dei dettagli a basso livello inerenti alle chiamate di sistema, così da evitare al comune programmatore di preoccuparsi di questo aspetto. Nel caso di Linux tale libreria è denominata *glibc*, una particolare implementazione della libreria standard C (*libc*).

[2]Particolare tool usato per testare e correggere programmi in fase di sviluppo

4.1.4 Driver di rete

Lo sviluppo di un device driver di rete è realizzato, sui sistemi GNU/Linux, attraverso una struttura definita appositamente per questo scopo denominata `net_device`. Essa racchiude tutte le funzioni necessarie per lo sviluppo di un device driver, oltre a fornire un'interfaccia che guida il programmatore nella definizione delle istruzioni da implementare. Lo scopo di `net_device` è infatti quello di modellizzare e riassumere tutti gli aspetti relativi ad un generica interfaccia di rete all'interno di un unica *struct*, in modo che siano di facile gestione. Per questo motivo `net_device` è una struttura estremamente eterogenea, che contiene informazioni di alto livello, quali per esempio dettagli sul protocollo di rete, assieme a dati di basso livello, relativi all'hardware fisico. Questo "difetto" rende `net_device` una struttura molto mutevole da una versione del kernel all'altra, a causa delle costanti modifiche apportate dagli sviluppatori, cosa che ha causato non pochi problemi in fase di porting del driver.

Coprire in maniera esauriente tutti i campi che compongono questa struttura esula da questa trattazione, quindi ci limiteremo a caratterizzarne solo gli aspetti direttamente interessati dallo sviluppo di Insane. Per una consultazione più approfondita rimandiamo alla relativa Appendice A.5, che riporta interamente il codice sorgente in oggetto.

Come già detto, `net_device` memorizza numerose informazioni di basso livello, quali per esempio l'indirizzo MAC del dispositivo (`dev_addr`) e le funzionalità offerte dall'hardware, organizzate in maniera binaria nella variabile `features`, unitamente a campi di carattere informativo, come il nome del device e le statistiche di trasmissione registrate.

È utile notare che dalla versione 2.6.30 del kernel è stato rimosso il puntatore relativo all'area privata del device, che quindi non può più essere utilizzata direttamente; è invece necessario ricorrere alla funzione `netdev_priv(dev)`, che fornisce un'implementazione più sicura ed efficiente, come spiegato in dettaglio nella sezione dedicata.

Infine, essendo `net_device` un'interfaccia, sono presenti i *prototipi* di tutte le funzioni principali che si eseguono su device di rete, quali operazioni sugli header dei pacchetti (racchiuse nella struttura `header_ops`), inizializzazione (`init`), trasmissione (`hard_start_xmit`), configurazione (`do_ioctl`) e rimozione (`stop`). È compito del programmatore, in fase di sviluppo del driver, fornire un'implementazione di queste funzioni ed associarle a quelle definite dall'interfaccia, in modo che siano riconosciute dal sistema operativo.

È significativo notare che questa parte è sostanzialmente ripetuta due volte, in quanto esiste una seconda struttura equivalente, chiamata `netdev_ops`, la quale può essere utilizzata in maniera simmetrica a quanto riportato sopra. Questa non è una ridondanza, quanto un esempio della natura "in divenire" che caratterizza `net_device`; gli sviluppatori stanno infatti portando a termine un processo di organizzazione delle funzioni, raggruppandole logicamente in strutture, cosa già effettuata per quanto riguarda `header_ops` e ancora in corso per `netdev_ops`. Dalla versione 2.6.31 sarà infatti obbligatorio usare `netdev_ops`, in quanto la controparte originale verrà rimossa; per questo motivo, si è scelto di abbandonare la compatibilità con la versione 2.6.29 (nella quale questa struttura non è ancora presente) in favore del nuovo metodo di dichiarazione, come spiegato nella sezione successiva.

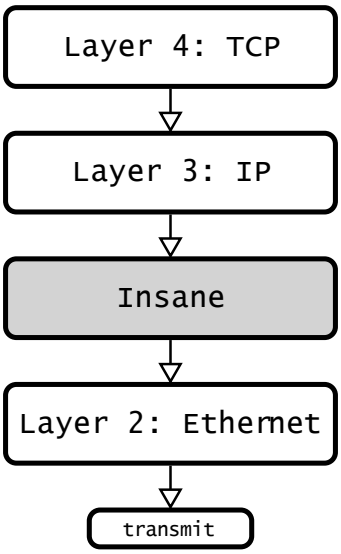

Figura 4.1: Posizione di Insane all'interno dello stack di rete

4.2 Il driver Insane versione 2.6

Lo sviluppo di Insane è partito dal lavoro messo a disposizione da Alessandro Rubini, che ha pubblicato un device driver in grado di realizzare un'interfaccia di rete dal comportamento *lossy*, ovvero con perdita controllata dei pacchetti, su un sistema GNU/Linux [19].

Insane agisce come un interfaccia *virtuale*; essa cioè non rappresenta un hardware fisicamente presente sulla macchina, come solitamente accade per i device di rete presenti sul sistema quali schede ethernet o wireless. Dal punto di vista del kernel, infatti, un'interfaccia non è altro che un modulo software in grado di processare pacchetti in uscita dal sistema, la cui implementazione rimane nascosta all'interno del driver. Questo permette di costruire particolari interfacce virtuali, utili per scopi specifici, delegando la trasmissione vera e propria a device secondari. Nasce in questo modo Insane, che sta per *INterface SAmple for Network Errors* (Figura 4.1).

Sfortunatamente, il driver originale, pensato e sviluppato sulla base del kernel 2.4, è inadatto ad un utilizzo sull'attuale versione 2.6, a causa della massiccia ristrutturazione che il codice sorgente ha subito relativamente al comparto networking. È stato quindi necessario un intenso lavoro di *porting*, in modo da riadattare l'implementazione originaria alla nuova interfaccia fornita dal kernel.

Questa procedura ha interessato principalmente le fasi di inizializzazione e registrazione del dispositivo, mentre invece la parte operativa dello stesso è rimasta sostanzialmente inalterata (non sono mancati però importanti aggiustamenti anche in questo senso).

Unitamente al driver, inoltre, è fornito un tool per la configurazione di Insane, denominato Insanely (Sezione 4.3). Essendo un programma userspace, che quindi non dipende direttamente dal sistema operativo, Insane non ha dovuto subire modifiche durante il porting, in quanto il suo modo di operare si appoggia su un'interfaccia standard (la chiamata di sistema `ioctl`) che è rimasta immutata da una versione all'altra.

4.2.1 Inizializzazione

La fase di inizializzazione è quella che ha richiesto più lavoro in fase di porting a causa della pesante modifica apportata all'architettura che gestisce l'impianto dei driver di rete. Essa viene lanciata durante l'init del modulo kernel, che avviene durante il caricamento dello stesso attraverso il comando `insmod`.

Innanzitutto si alloca, mediante l'istruzione `alloc_netdev()`, lo spazio necessario a memorizzare i dati e le strutture proprie del driver, e in particolare la sua parte privata, di cui deve essere specificata la dimensione come parametro. È in questa fase che la funzione `insane_init()` viene richiamata per inizializzare i rimanenti campi della struttura `net_device`; è necessario infatti dichiarare esplicitamente l'associazione fra le istruzioni utilizzate localmente e le corrispondenti definizioni nell'interfaccia kernel, operazione che viene eseguita nel seguente modo:

```
.ndo_open         = insane_open,
.ndo_stop         = insane_close,
.ndo_do_ioctl     = insane_ioctl,
.ndo_get_stats    = insane_get_stats,
.ndo_start_xmit   = insane_xmit,
.ndo_neigh_setup  = insane_neigh_setup_dev,
```

I restanti campi vengono delegati alla funzione `ether_setup(dev)`, che si occupa di configurare i parametri dell'interfaccia virtuale come un dispositivo ethernet. Infine, è importante assegnare un indirizzo MAC al device, in questo caso scelto in maniera casuale attraverso `random_ether_addr()`; l'operazione, se non eseguita, inficerebbe il successivo funzionamento del software. Terminata la preparazione del dispositivo, è infine possibile renderlo riconoscibile al resto del sistema, cosa che avviene registrando il device con l'istruzione `register_netdev()`.

Da notare che in questa fase Insane non è ancora del tutto attivo: per terminare la configurazione è necessario utilizzare il tool `ifconfig`, assegnando al device lo stesso indirizzo IP dell'interfaccia associata. Così facendo si richiama la funzione `insane_open()`, la quale si occupa di avviare la coda di pacchetti attraverso l'istruzione `netif_start_queue()`.

4.2.2 Zona privata

Ogni device driver ha la necessità di memorizzare particolari informazioni necessarie al proprio funzionamento, come per esempio parametri e statistiche. Dato che non è possibile progettare a priori strutture predefinite per la memorizzazione di tali dati, gli sviluppatori hanno creato una particolare struttura, chiamata `netdev_priv`, la quale può essere personalizzata a piacimento dall'ideatore del driver.

Nel caso di Insane, tale struttura memorizza cinque informazioni fondamentali: un puntatore ad una struttura `device_stats`, che mantiene le statistiche dell'interfaccia (pacchetti trasmessi, pacchetti persi, errori, ecc...), un puntatore al device di rete incaricato dell'effettiva trasmissione dei pacchetti (chiamato `priv_device`) e tre parametri numerici di tipo intero, che rappresentano rispettivamente la modalità di perdita dei pacchetti (`priv_mode`) e i parametri che la caratterizzano (`priv_arg1` e `priv_arg2`).

L'accesso a questa struttura è critico dal punto di vista delle performance e della sicurezza, dato che viene eseguito frequentemente in numerose funzioni del driver.

Per questo motivo, nelle recenti versioni del kernel, gli sviluppatori sconsigliano l'accesso diretto alla struttura con un istruzione del tipo:

```
struct insane_private *priv = insane_dev.priv;
```

cosa che veniva invece regolarmente fatta nella vecchia versione di Insane.

Nella nuova versione questo aspetto è stato corretto, sostituendo le chiamate sopraccitate con la funzione `netdev_priv(dev)`, appositamente ideata per restituire la struttura privata in maniera sicura ed efficiente tramite l'uso dell'aritmetica dei puntatori. L'uso di questo metodo è stato caldamente raccomandato fino alla versione 2.6.29 del kernel ed è diventato obbligatorio nell'ultima versione, la 2.6.30.

4.2.3 Trasmissione

Il nucleo funzionale di Insane è costituito dalla sezione di trasmissione, implementata dall'istruzione `insane_xmit()`, nella quale viene effettivamente realizzata la perdita controllata di pacchetti.

Insane offre due modelli di funzionamento: è possibile scegliere di inoltrare una percentuale prefissata del traffico di rete in transito, ideale per simulare un canale rumoroso nel quale, in maniera casuale, un numero pressoché costante di frame ogni cento viene perduto. Inoltre è disponibile una modalità a tempo, con la quale vengono alternativamente scartati e trasmessi pacchetti per intervalli temporali selezionabili dall'utente. Queste due modalità sono rispettivamente identificate dai valori `INSANE_PERCENT` e `INSANE_TIME` (oltre a `INSANE_PASS`, che disattiva il comportamento lossy del driver), memorizzate nella variabile `priv_mode` della zona privata.

In caso di modalità percentuale (oggetto dei test seguenti), viene creato un valore pseudo-casuale compreso fra 0 e 100 tramite un generatore semplice ma in grado di offrire una buona imprevedibilità e distribuzione dei risultati [20]:

```
if (!randval) randval = jiffies;
randval = ((randval * 1103515245) + 12345) & 0x7fffffff;
accept = (randval % 100) < priv->priv_arg1;
```

La variabile `jiffies` è direttamente proporzionale al tempo di uptime del sistema utilizzato, quindi in grado di garantire una buona casualità per essere usata come *seed*[3] per il generatore. In seguito, il risultato viene confrontato con la percentuale di perdita specificata dall'utente (presente in `priv_arg1`), determinando se il pacchetto corrente deve essere inviato o meno.

Nella modalità a tempo, invece, il controllo è più semplice e prevede solamente la verifica se l'istante corrente è contenuto nell'intervallo di tempo nel quale è ammessa la trasmissione:

```
randval = jiffies % (priv->priv_arg1 + priv->priv_arg2);
accept = randval < priv->priv_arg1;
```

In entrambe le situazioni, la variabile `accept` contiene l'esito del pacchetto: nel caso rappresenti il valore *vero*, si procede con la trasmissione dello stesso tramite l'istruzione `dev_queue_xmit(skb)` e il successivo aggiornamento delle statistiche; in caso contrario i dati vengono scartati.

[3] Valore utilizzato per inizializzare un generatore di numeri pseudo-casuale

Quest'ultima fase ha visto un'importante modifica in fase di porting, con l'utilizzo dell'istruzione `kfree_skb(skb)`, che libera la porzione di memoria utilizzata dal pacchetto inutilizzato. Precedentemente questo aspetto veniva ignorato, portando ad occupazioni eccessive di RAM (*memory leak*) e il susseguente blocco del sistema.

4.2.4 Ioctl

Una delle system call più importanti offerte dal sistema operativo è le cosiddetta funzione di Controllo Input/Output o, in breve, `ioctl`. Essa fa parte dell'impianto software che instaura un ponte fra programmi utente e kernel, ed in particolare è usata per la configurazione dei device driver, ognuno dei quali può supportare ioctl di lettura (per inviare informazioni dal processo utente al kernel), scrittura (per ritornare dati all'utente), o entrambi.

La funzione ioctl accetta tre parametri, che specificano rispettivamente il *file descriptor* del driver associato, il codice della richiesta ioctl, e un parametro, che di fatto informa il driver su cosa fare.

Insane dispone di un comparto ioctl semplice ed essenziale, composto da due modalità. Il comando `SIOCINSANESETINFO` permette di settare i parametri di funzionamento, nello specifico l'interfaccia di rete associata, la modalità di perdita dei pacchetti ed eventuali parametri, informazioni che verranno poi memorizzate nell'area privata del driver. Esiste inoltre la controparte di lettura `SIOCINSANEGETINFO`, che viene impiegata dal tool di configurazione Insanely per visualizzare i settaggi correnti di Insane, nel caso venga lanciato senza parametri.

Da notare che il passaggio di dati fra utente e driver deve essere implementato tramite funzioni particolari, chiamate `copy_from_user()` e `copy_to_user()`, che realizzano il trasferimento dei parametri tra User Space e Kernel Space, come mostrato in figura 4.2.

4.3 Tool di configurazione

Come già introdotto in precedenza, l'interfaccia virtuale può essere configurata rapidamente tramite la chiamata di sistema `ioctl`, la quale si occupa di inoltrare i parametri passati dall'utente allo spazio kernel. Ciò è reso possibile grazie ad un software userspace chiamato *Insanely*, il cui compito consiste nel ricevere i dati di configurazione, formattandoli correttamente in una richiesta che può essere compresa da Insane.

Il vettore di informazioni è rappresentato da `ifreq`, la cui struttura offre un'interfaccia standard indipendente dall'hardware specifico e dalle sue funzionalità. All'interno sono presenti alcuni campi utili a definire i dettagli della richiesta, quali il nome del destinatario (`ifrn_name`) ed eventuali indirizzi o flag (`ifru_flags`).

Nel caso in cui l'oggetto della richiesta siano parametri dotati di un'impostazione particolare, è presente un puntatore generico, denominato `ifru_data`, che l'utente può assegnare a piacimento ad una qualsiasi struttura definita in precedenza. Insanely ricorre a questo meccanismo, incapsulando i parametri da inoltrare in una `struct` secondaria di tipo `insane_userinfo`, che ricalca l'organizzazione della zona privata del driver:

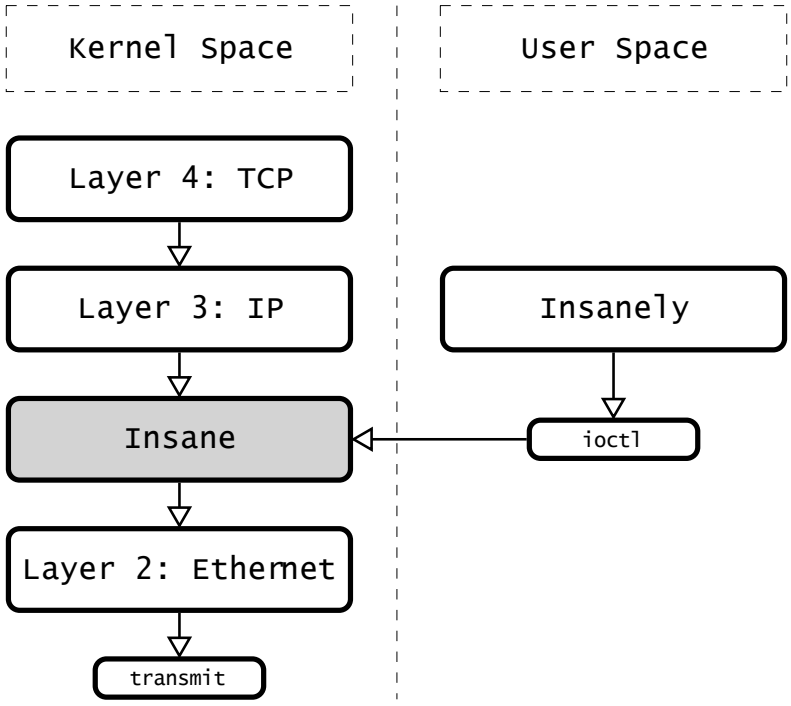

Figura 4.2: Meccanismo di configurazione di Insane

```
struct insane_userinfo {
        char name[INSANE_NAMELEN];
        int mode;
        int arg1;
        int arg2;
};
```

Una volta effettuati i dovuti controlli sulla coerenza dei dati in ingresso e sul corretto funzionamento di Insane, il software costruisce le strutture precedentemente citate contenenti le informazioni specificate dall'utente, che verranno poi passate allo spazio kernel. L'effettivo inoltro delle richieste `ioctl` avviene per mezzo di un *socket*[4] appositamente creato:

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Esso viene poi specificato come argomento dell'istruzione seguente, assieme a `SIOCINSANESETINFO` (utile al driver per identificare il tipo di richiesta) e al puntatore alla struttura contenente i parametri in oggetto (Figura 4.3):

```
ioctl(sock, SIOCINSANESETINFO, &req)
```

Una volta giunta nello spazio kernel, la richiesta viene presa in carico da Insane, che ne ricava la modalità di funzionamento specificata e l'hardware di rete a cui deve associarsi. È significativo notare che in questa fase il driver "clona" alcune proprietà, quali indirizzo fisico e di broadcasting, dell'interfaccia figlia:

```
memcpy(dev->dev_addr,  slave->dev_addr,  sizeof(...));
memcpy(dev->broadcast, slave->broadcast, sizeof(...));
```

[4]Punto di accesso del codice applicativo allo stack di rete

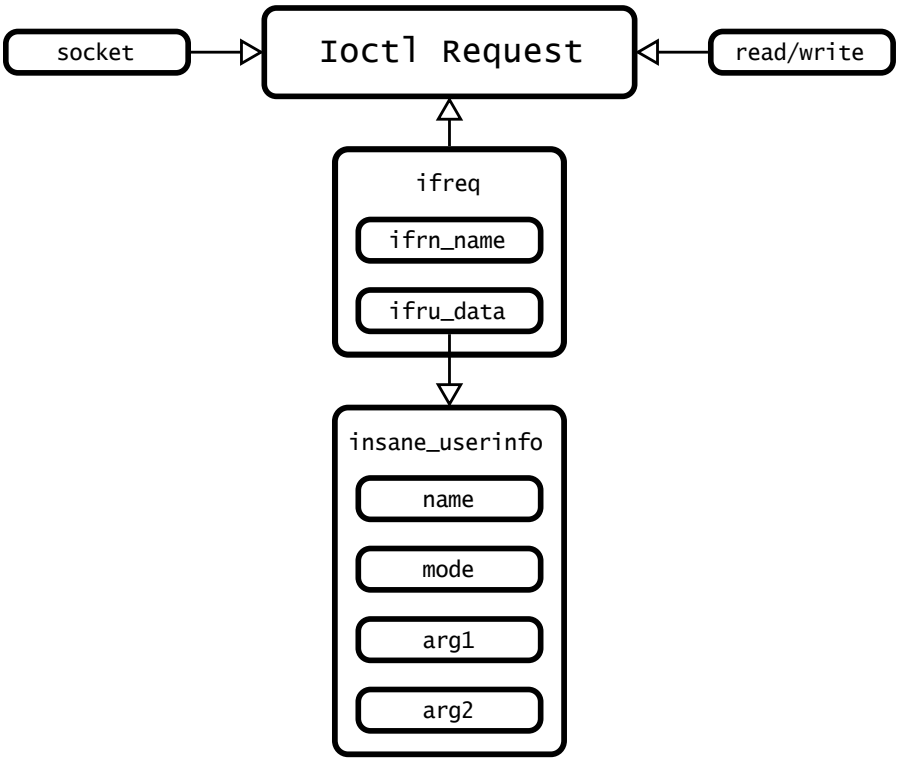

Figura 4.3: Struttura di una richiesta `ioctl`

Infine, Insanely offre la possibilità di visualizzare modalità di funzionamento corrente di Insane, mediante una `ioctl` di lettura, operante in modo sostanzialmente inverso a quanto appena illustrato. La differenza fondamentale, oltre all'uso del codice `SIOCINSANEGETINFO` come parametro, sta nel fatto che, in questo caso, il driver esegue un operazione di scrittura, e non di lettura, sulla struttura `insane_userinfo` mediante l'istruzione `copy_to_user()`. Per questo motivo è fondamentale che l'indirizzo contenuto nel campo `ifru_data` sia riferito allo spazio utente, come da specifica per l'istruzione sopraccitata.

4.4 Routing

Una volta installata e configurata l'interfaccia virtuale, sono necessari alcuni ulteriori passaggi per permetterle di funzionare correttamente. Essa infatti non è ancora dotata di un indirizzo IP, il che rende impossibile la corretta trasmissione di pacchetti, dato che non esiste una regola di routing che inoltra del traffico di rete verso di essa.

Per connettere Insane alla rete locale, è quindi necessario assegnargli lo stesso IP della scheda ethernet su cui si appoggia, per permettere agli host remoti di rispondere correttamente alle richieste inviate:

```
ifconfig insane 192.168.2.1
```

In questo modo il traffico in uscita verso la sottorete corrispondente sarà soggetto alla perdita controllata realizzata da Insane; da notare che, per come è implementata la trasmissione, solo i pacchetti inviati saranno interessati dal fenomeno, mentre le

risposte manterranno le funzionalità usuali, a meno che non si installi un'istanza di Insane anche sui restanti nodi di rete.

In determinati casi può essere utile restringere tale comportamento solo verso alcuni host, per esempio per effettuare alcuni test comparativi, come vedremo nel Capitolo 5. È quindi necessario correggere le regole di instradamento in modo da ridirigere solamente una parte del traffico verso verso Insane e inoltrare direttamente al dispositivo ethernet i pacchetti rimanenti (Figura 4.4).

```
route del -net 192.168.2.0/24 dev insane
route add -host 192.168.2.1 dev insane
```

Nell'esempio sopraccitato si constaterà una perdita di pacchetti solamente verso l'host 192.168.2.1, mentre il traffico verso la restante parte di sottorete non sarà interessato.

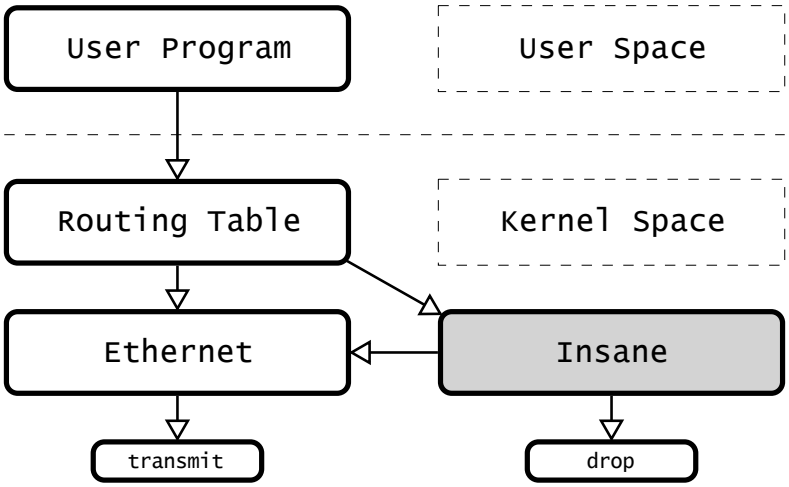

Figura 4.4: Relazione fra Insane e regole di routing

Capitolo 5

Test comparativi

Per constatare l'effettivo funzionamento del driver Insane, è stato necessario effettuare una serie di test comparativi atti a verificarne il comportamento e le prestazioni. Insane è stato sottoposto ad una serie di prove mediante alcuni tool standard nell'analisi delle prestazioni di rete, confrontando i risultati con quelli fatti registrare da *Iptables* e *Netem*, che, come esposto nella Sezione 3.4, sono in grado di simulare una perdita controllata di pacchetti. Infine, è stato valutato il consumo di risorse da parte dei tre programmi, verificando in che misura la loro presenza incide sul carico computazionale della macchina.

5.1 Ambiente di test

I vari test sono stati condotti su una piccola rete ad-hoc, costituita da due end-host connessi ad altrettante subnet differenti. Fra di esse è stato installato un nodo router, sul quale sono stati via via attivati i vari meccanismi di perdita controllata sopraccitati (Figura 5.1). Le macchine sono state inoltre dotate di sistema operativo Ubuntu versione Server, una distribuzione appositamente pensata per essere impiegata su nodi di rete. Come modalità riferimento è stata considerata una perdita percentuale del 50%, realizzata in maniera equivalente con i tre tool a disposizione, verificando in seguito i parametri statistici della rete, quali latenze e velocità di trasferimento. Questi due aspetti dell'analisi prestazionale sono stati calcolati mediante l'uso di due software specifici, *Ping* e *Iperf*.

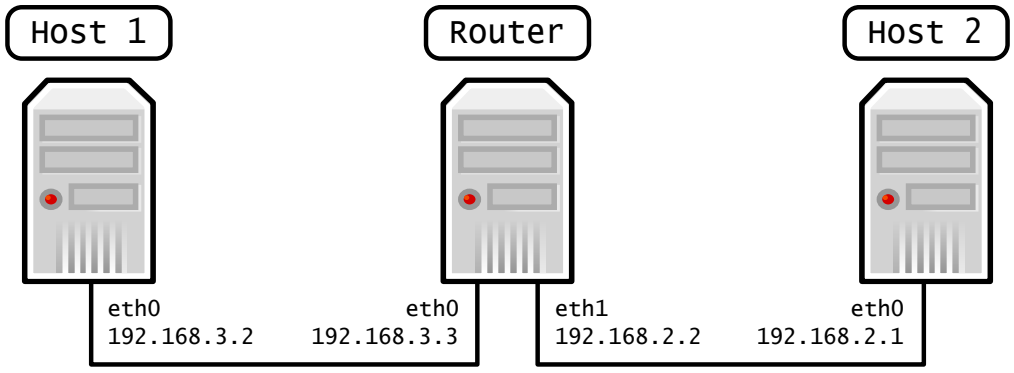

Figura 5.1: Topologia della rete utilizzata per effettuare i test

5.2 Ritardo

Il software *Ping* è senza dubbio uno dei più utilizzati nel campo delle comunicazioni digitali, in quanto fornisce un metodo rapido e semplice per verificare il funzionamento di uno o più nodi di rete. Esso si appoggia sul protocollo ICMP (*Internet Control Message Protocol*) [21], utilizzato principalmente per inviare messaggi informativi o di errore indicanti lo stato di funzionamento di un determinato host. Per questo motivo la struttura del protocollo è estremamente semplice, rendendolo adatto ad essere impiegato per gli scopi sopraccitati.

Il programma si basa su un principio tanto semplice quanto efficace: inviare un pacchetto di richiesta e misurare il tempo che trascorre fino alla ricezione della risposta corrispondente. Il parametro così determinato è definito *Round Trip Time* o *Rtt*, il cui valore gioca un ruolo fondamentale nell'analisi prestazionale di una rete, determinando il ritardo di trasmissione lungo il cammino.

In Tabella 5.1 sono riportati i valori riscontrati con un test di 100 pacchetti, eseguito confrontando i risultati dei tre software con un test di riferimento a rete scarica. Il valore di Rtt medio rilevato è uniforme, e si assesta costantemente attorno ad un valore di 0.155 ms, sia nel risultato di riferimento, sia in presenza di perdita controllata (Figura 5.2).

Tabella 5.1: Valori di Round Trip Time misurati

	Rtt_{MIN}	Rtt_{AVG}	Rtt_{MAX}	Rtt_{MDEV}
Nessuno	0.149 ms	0.156 ms	0.257 ms	0.012 ms
Insane	0.150 ms	0.159 ms	0.188 ms	0.011 ms
Iptables	0.146 ms	0.155 ms	0.216 ms	0.019 ms
Netem	0.153 ms	0.157 ms	0.168 ms	0.017 ms

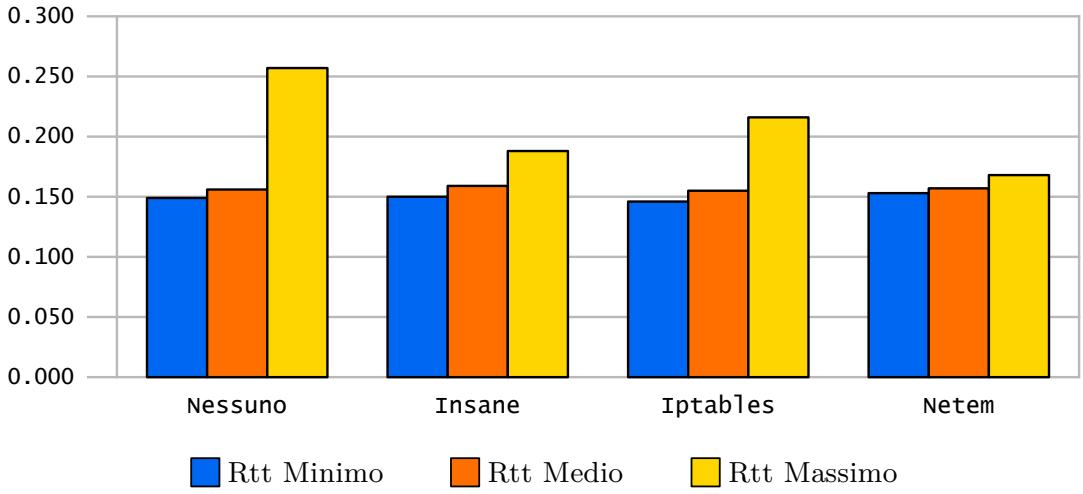

Figura 5.2: Grafico comparativo dei valori di Round Trip Time

Come è logico aspettarsi, il valore di questo parametro non varia in presenza di perdita o meno, in quanto è calcolato solamente sulla base delle risposte correttamente ricevute, mentre i pacchetti perduti vengono ovviamente ignorati e non entrano a far parte del pool di dati che determina i valori statistici.

Infine, dall'analisi del fenomeno del jitter (Figura 5.3) emerge come esso abbia un andamento discontinuo, facendo registrare un valore elevatissimo utilizzando Iptables, anomalia non presente invece nei software concorrenti, che si attestano attorno ad una misura di 0.45 ms (Tabella 5.2). Questo comportamento insolito è da ricondurre al metodo di funzionamento del programma, che, al contrario delle soluzioni a confronto, lavora nello spazio di indirizzamento utente, causando così un numero elevato di cambi di contesto con la conseguente generazione di latenze elevate.

Tabella 5.2: Valori di Jitter misurati

	Jitter
Nessuno	0.423 ms
Insane	0.449 ms
Iptables	30.826 ms
Netem	0.460 ms

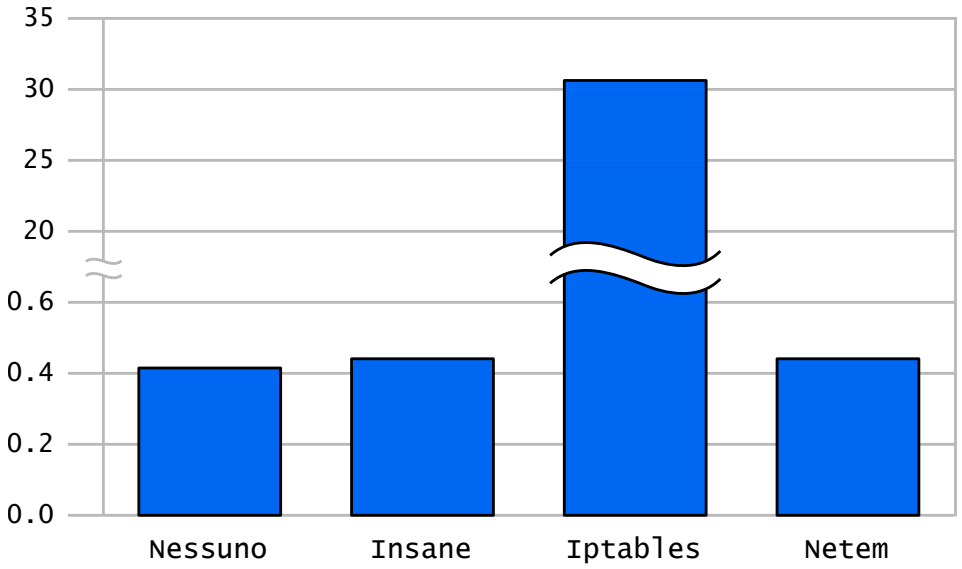

Figura 5.3: Valori di jitter rilevati da Iperf

5.3 Banda

Per la stima della capacità di trasmissione della rete è stato utilizzato il programma *Iperf* [22], in grado di effettuare un'analisi prestazionale e mostrare le reali velocità raggiungibili mediante la simulazione di un trasferimento di file fra due host.

In particolare, Iperf genera un traffico UDP a bitrate costante per una durata di tempo configurabile, durante la quale vengono tenute statistiche quali la banda disponibile, il tasso di perdita dei pacchetti, il traffico generato e il jitter riscontrato. Per questo motivo, il programma è strutturato secondo il modello client-server: il client è responsabile della generazione del *data stream* verso il server, al quale sono affidati i compiti di misurazione sopraccitati.

In fase di test, il server e il client sono stati rispettivamente avviati tramite i seguenti comandi:

```
iperf -u -s -U -i 30 -p 5005
iperf -u -c 192.168.2.1 -p 5005 -i 30 -t 60 -b 100M
```

Con queste direttive, è stato generato un traffico UDP alla velocità di 100 MB/s per un tempo totale di 60 secondi, durante i quali vengono registrate due misurazioni, una ogni 30 secondi.

Il test così effettuato ha dato i risultati riportati in Tabella 5.3, in cui si nota come la capacità di rete iniziale di 77.7 MB/s si riduca a circa 38.7 MB/s in tutti i risultati.

Tabella 5.3: Analisi prestazionale effettuata con Iperf

	Perdita	Dati Trasferiti	Velocità
Nessuno	0%	556 MB	77.7 MB/s
Insane	50%	277 MB	38.7 MB/s
Iptables	50%	277 MB	38.4 MB/s
Netem	50%	277 MB	38.7 MB/s

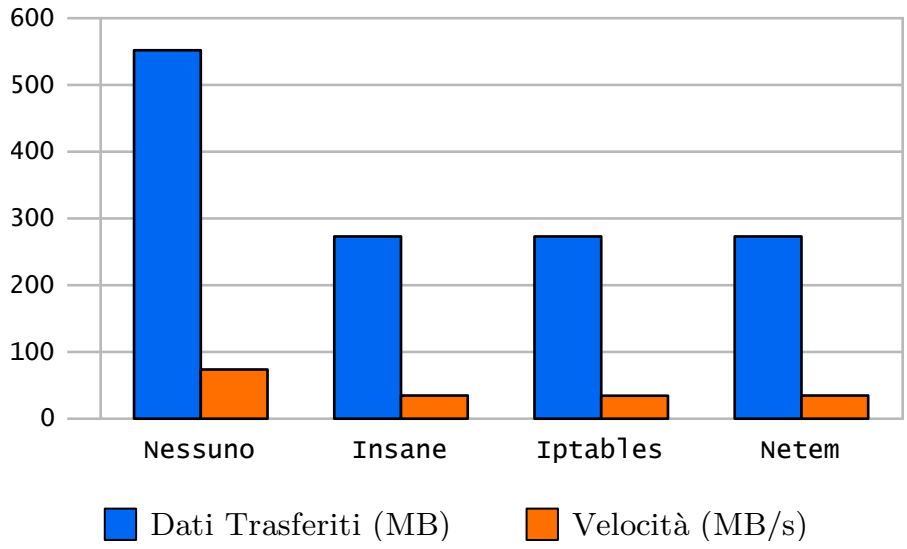

Figura 5.4: Traffico inviato e velocità di trasferimento rilevati da Iperf

Dai risultati dei test si possono trarre alcune importanti conclusioni: innanzitutto si nota come la quantità di dati correttamente ricevuti dal server è effettivamente la metà del totale inviato dal client (Figura 5.4), segnale che la perdita di pacchetti funziona come pronosticato.

È inoltre confermata la pesante influenza di tale fenomeno sulle prestazioni della rete: si può verificare facilmente come la capacità di trasferimento sia dimezzata, in quanto per inviare correttamente la stessa mole di dati al server è necessario il doppio del tempo.

5.4 Richieste hardware

In termini di risorse richieste, i tre software hanno fatto registrare un comportamento del tutto equivalente, influendo in maniera trascurabile sul carico computazionale della macchina, sia in termini di memoria Ram allocata, sia per quanto riguarda l'occupazione della Cpu. L'unico effetto apprezzabile è un aumento dello 0,3% dell'attività del processore, che, essendo presente anche nel test di riferimento (Tabella 5.4), è da attribuire alla procedura di trasmissione dei pacchetti, piuttosto che alla loro perdita da parte dei software in esame.

Tabella 5.4: Risorse hardware richieste

	Ram	Cpu
Nessuno	~0	0,3%
Insane	~0	0,3%
Iptables	~0	0,3%
Netem	~0	0,3%

5.5 Tasso di perdita

Dai test appena esposti emerge come tutti i software confrontati realizzino correttamente il fenomeno della perdita controllata di pacchetti, mostrandosi sostanzialmente equivalenti dal questo punto di vista.

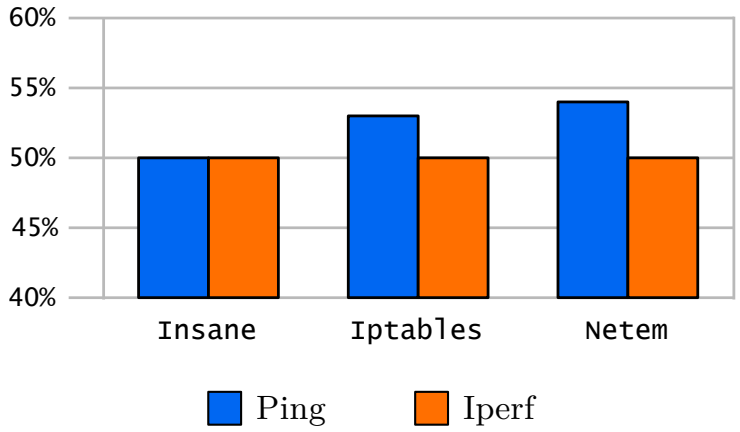

Figura 5.5: Tasso reale di perdita dei pacchetti (teorico 50%)

Possiamo inoltre notare come Insane faccia registrare una buona precisione anche con un numero di pacchetti inviati relativamente basso (nell'ordine del centinaio) in occasione del test con Ping (Figura 5.5). Infatti, mentre con Netem si ha un tasso di perdita del 54%, Iptables scarta invece il 53% del traffico; le ottime prestazioni di Insane, che perde esattamente metà dei pacchetti sono dovute alla bontà dell'algoritmo di generazione dei numeri pseudo-casuali utilizzato.

Capitolo 6

Conclusioni

"Non so come il mondo potrà giudicarmi ma a me sembra soltanto di essere un bambino che gioca sulla spiaggia, e di essermi divertito a trovare ogni tanto un sasso o una conchiglia più bella del solito, mentre l'oceano della verità giaceva insondato davanti a me"

Isaac Newton

Lo studio della perdita di pacchetti è senza dubbio di vitale importanza nell'ambito delle reti digitali e, come si è visto, è essenziale disporre di strumenti di test adeguati per il miglioramento del software e delle infrastrutture di comunicazione.

Anche se al giorno d'oggi la maggior parte dei link di trasmissione risulta molto affidabile, rimangono comunque settori in cui il fenomeno si ripropone marcatamente, quali per esempio le reti wireless, che stanno percorrendo una fase di enorme diffusione. Come mostrato dai test sperimentali, infatti, la perdita di pacchetti è in grado di influire pesantemente sulle prestazione della rete, riducendo notevolmente la capacità di trasmissione e introducendo latenze elevate.

Uno studio accurato del fenomeno è quindi essenziale, ed in questo contesto il lavoro presentato acquista un valore particolare, grazie al marcato approccio sperimentale, che ha permesso di ottenere risultati molto validi e di qualità superiore rispetto a quelli ottenibili, per esempio, con un simulatore di rete. Sempre più router adottano Linux come sistema operativo, grazie alla possibilità di ottenere distribuzioni estremamente efficienti e leggere, pensate appositamente per l'ambito delle telecomunicazioni. Uno studio che coinvolge questo sistema operativo è quindi particolarmente indicato, dato che ricrea una situazione effettivamente presente nei contesti reali.

Inoltre, grazie all'intenso lavoro di *porting* effettuato, è stato possibile rimodernare e rendere operativo un ottimo progetto, l'interfaccia Insane, il cui sviluppo ha portato ad un notevole approfondimento nell'ambito del kernel Linux e delle sue continue innovazioni introdotte versione dopo versione. Dai test è infatti emerso come le prestazioni del device driver siano molto buone, mostrando un'ottima funzionalità e delle richieste hardware molto esigue, in linea o in alcuni casi migliori dei software concorrenti.

I promettenti risultati ottenuti danno lo spunto per un'ulteriore sviluppo, atto a migliorare ed espandere quanto è già implementato: è infatti prospettabile l'aggiunta di nuove funzionalità al driver (modelli di perdita aggiuntivi o più parametrizzati, simulazione di latenze più o meno elevate, ecc...) oppure la sua inclusione nella sezione di test del kernel Linux.

Lo studio, insomma, è tutt'altro che concluso, e lascia spazio ad ampi sviluppi sotto molteplici aspetti. La capillare diffusione che Linux sta vivendo in moltissimi settori fornisce un valore aggiunto ed, allo stesso tempo, un punto di partenza per evoluzioni future nella direzione mostrata da questo elaborato.

Appendice A

Codice Sorgente

A.1 insane.h

```
1  #ifndef __INSANE_H__
2  #define __INSANE_H__
3
4  enum insane_mode {
5      INSANE_PASS = 0,    /* every packet is transmitted (default) */
6      INSANE_PERCENT,     /* pass some percent of the packets */
7      INSANE_TIME,        /* work (and fail) on a timely basis */
8  };
9
10 #define INSANE_NAMELEN 16
11
12 /* Structure used to exchange data during ioctl commands */
13 struct insane_userinfo {
14     char name[INSANE_NAMELEN];
15     int mode;
16     int arg1;
17     int arg2;
18 };
19
20 /* These are the two ioctl commands needed to interact with insane */
21 #define SIOCINSANESETINFO SIOCDEVPRIVATE
22 #define SIOCINSANEGETINFO (SIOCDEVPRIVATE+1)
23
24 #endif /* __INSANE_H__ */
```

A.2 insane.c

```
1  /*
2   * insane.c --  source for the "insane" module, a sample virtual interface
3   *
4   * Copyright (c) 2000 Alessandro Rubini (rubini@linux.it)
5   *
6   *   This program is free software; you can redistribute it and/or modify
7   *   it under the terms of the GNU General Public License as published by
8   *   the Free Software Foundation; either version 2 of the License, or
9   *   (at your option) any later version.
10  *
11  *   This program is distributed in the hope that it will be useful,
12  *   but WITHOUT ANY WARRANTY; without even the implied warranty of
13  *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  *   GNU General Public License for more details.
15  *
16  *   You should have received a copy of the GNU General Public License
17  *   along with this program; if not, write to the Free Software
18  *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
19  */
```

```
20 #ifndef __KERNEL__
21 #  define __KERNEL__
22 #endif
23
24 /* The Makefile takes care of adding -DMODULE */
25
26 #include <linux/etherdevice.h>
27 #include <linux/module.h>
28
29 #include <linux/kernel.h>       /* printk() */
30 #include <linux/slab.h>         /* kmalloc() */
31 #include <linux/errno.h>        /* error codes */
32 #include <linux/netdevice.h>    /* basic data structures */
33 #include <linux/init.h>         /* __init */
34 #include <linux/if_arp.h>       /* ARPHRD_ETHER */
35 #include <linux/version.h>      /* LINUX_VERSION_CODE */
36 #include <net/arp.h>            /* neighbor stuff */
37 #include <asm/uaccess.h>        /* memcpy and such */
38
39 #include "insane.h"
40
41 #define DRIVER_AUTHOR           "Port: Filippo Bistaffa <liquidator87@gmail.com>"
42                                 // Source: Alessandro Rubini <rubini@linux.it>
43 #define DRIVER_DESC             "Insane Driver"
44
45 MODULE_LICENSE("GPL");
46 MODULE_AUTHOR(DRIVER_AUTHOR);
47 MODULE_DESCRIPTION(DRIVER_DESC);
48
49 struct net_device *insane_dev;
50
51 /* ------------------------------------------------------------------------
52  * definition of the "private" data structure used by this interface
53  */
54 struct insane_private {
55     struct net_device_stats priv_stats;
56     struct net_device *priv_device; /* interface used to xmit data */
57     int priv_mode; /* how to drop packets */
58     int priv_arg1; /* arguments to the dropping mode */
59     int priv_arg2;
60 };
61
62 /* ------------------------------------------------------------------------
63  * open and close
64  */
65 int insane_open(struct net_device *dev)
66 {
67     /* mark the device as operational */
68     printk("%s: device opened\n", dev->name);
69     netif_start_queue(dev);
70     return 0;
71 }
72
73 int insane_close(struct net_device *dev)
74 {
75     printk("%s: device closed\n", dev->name);
76     netif_stop_queue(dev);
77     return 0;
78 }
79
80 /* ------------------------------------------------------------------------
81  * get_stats: return a pointer to the device statistics
82  */
83 struct net_device_stats *insane_get_stats(struct net_device *dev)
84 {
85     return &((struct insane_private *)netdev_priv(dev))->priv_stats;
86 }
87
88 /* ------------------------------------------------------------------------
89  * header stuff: fall back on the slave interface to deal with this stuff
90  */
```

```
91  static int insane_create_header(struct sk_buff *skb, struct net_device *dev,
92          unsigned short type, const void *daddr, const void *saddr, unsigned len)
93  {
94      struct insane_private *priv = netdev_priv(insane_dev);
95      int retval;
96
97      skb->dev = priv->priv_device;
98      retval = skb->dev->header_ops->create(skb, skb->dev, type,
99                                            daddr, saddr, len);
100     skb->dev = dev;
101     return retval;
102 }
103
104 static int insane_rebuild_header(struct sk_buff *skb)
105 {
106     struct insane_private *priv = netdev_priv(insane_dev);
107     int retval;
108
109     skb->dev = priv->priv_device;
110     retval = skb->dev->header_ops->rebuild(skb);
111     skb->dev = insane_dev;
112     return retval;
113 }
114
115 /* ------------------------------------------------------------------------
116  * create default header_ops struct
117  */
118 static const struct header_ops insane_header_ops = {
119     .create  = insane_create_header,
120     .rebuild = insane_rebuild_header,
121     .cache   = NULL, /* disable caching */
122 };
123
124 /* ------------------------------------------------------------------------
125  * neighbors: this comes form shaper.c (Alan Cox) and is needed for ARP to work
126  */
127 int insane_neigh_setup(struct neighbour *n)
128 {
129     if (n->nud_state == NUD_NONE) {
130         n->ops = &arp_broken_ops;
131         n->output = n->ops->output;
132     }
133     return 0;
134 }
135
136 int insane_neigh_setup_dev(struct net_device *dev, struct neigh_parms *p)
137 {
138     if (p->tbl->family == AF_INET) {
139         p->neigh_setup = insane_neigh_setup;
140         p->ucast_probes = 0;
141         p->mcast_probes = 0;
142     }
143     return 0;
144 }
145
146 /* ------------------------------------------------------------------------
147  * xmit: actual delivery (or not) of the data packets
148  */
149 int insane_xmit(struct sk_buff *skb, struct net_device *dev)
150 {
151     struct insane_private *priv = netdev_priv(dev);
152     int accept; /* accept this packet or drop it */
153     static unsigned long randval;
154
155     if (!priv->priv_device) {
156         /* cannot send to anyone, just return */
157         kfree_skb(skb);
158         priv->priv_stats.tx_errors++;
159         priv->priv_stats.tx_dropped++;
160         return 0;
161     }
```

```
162     switch (priv->priv_mode) {
163
164         case INSANE_PERCENT:
165             if (!randval) randval = jiffies; /* a typical seed */
166             /* hash the value, according to the TYPE_0 rule of glibc */
167             randval = ((randval * 1103515245) + 12345) & 0x7fffffff;
168             accept = (randval % 100) < priv->priv_arg1;
169             break;
170
171         case INSANE_TIME:
172             randval = jiffies % (priv->priv_arg1 + priv->priv_arg2);
173             accept = randval < priv->priv_arg1;
174             break;
175
176         case INSANE_PASS:
177         default: /* unknown mode: default to pass */
178             accept = 1;
179     }
180
181     if (!accept) {
182         kfree_skb(skb);
183         priv->priv_stats.tx_errors++;
184         priv->priv_stats.tx_dropped++;
185         return 0;
186     }
187     /* else, pass it to the real interface */
188
189     priv->priv_stats.tx_packets++;
190     priv->priv_stats.tx_bytes += skb->len;
191
192 #if 0
193     return priv->priv_device->hard_start_xmit(skb, priv->priv_device);
194 #else
195     skb->dev = priv->priv_device;
196     skb->priority = 1;
197     dev_queue_xmit (skb);
198     return 0;
199 #endif
200 }
201
202 /* ----------------------------------------------------------------------------
203  * ioctl: let user programs configure this interface
204  */
205 int insane_ioctl(struct net_device *dev, struct ifreq *ifr, int cmd)
206 {
207     int err;
208
209     struct net_device *slave;
210     struct insane_private *priv = netdev_priv(dev);
211     /* hold a local (kernel-space) copy of the configuration data */
212     struct insane_userinfo info;
213     /* and a pointer into user space as well */
214     struct insane_userinfo *uptr = (struct insane_userinfo *)ifr->ifr_data;
215
216     /* only authorized users can control the interface */
217     if (cmd == SIOCINSANESETINFO && !capable(CAP_NET_ADMIN))
218         return -EPERM;
219
220     /* process the command */
221     switch(cmd) {
222         case SIOCINSANEGETINFO: /* return configuration to user space */
223
224             /* interface name */
225             memset(info.name, 0, INSANE_NAMELEN);
226             if (priv->priv_device)
227                 strncpy(info.name, priv->priv_device->name, INSANE_NAMELEN-1);
228
229             /* parameters */
230             info.mode = priv->priv_mode;
231             info.arg1 = priv->priv_arg1;
232             info.arg2 = priv->priv_arg2;
```

```
233             /* return the data structure to  user space */
234             err = copy_to_user(uptr, &info, sizeof(info));
235             if (err) return err;
236             break;
237 
238         case SIOCINSANESETINFO:
239 
240             /* retrieve the data structure from user space */
241             err = copy_from_user(&info, uptr, sizeof(info));
242             if (err) return err;
243 
244             printk("name: %s, arg %i %i\n", info.name, info.arg1, info.arg2);
245 
246             /* interface name */
247             slave = __dev_get_by_name(&init_net, info.name);
248             if (!slave)
249                 return -ENODEV;
250             if (slave->type != ARPHRD_ETHER && slave->type != ARPHRD_LOOPBACK)
251                 return -EINVAL;
252 
253             /* The interface is good, get hold of it */
254             priv->priv_device = slave;
255             if (slave->header_ops)
256                 dev->header_ops = &insane_header_ops;
257             else
258                 dev->header_ops = NULL;
259 
260             /* also, and clone its IP, MAC and other information */
261             memcpy(dev->dev_addr,  slave->dev_addr,  sizeof(slave->dev_addr));
262             memcpy(dev->broadcast, slave->broadcast, sizeof(slave->broadcast));
263 
264             /* accept the parameters (no checks here) */
265             priv->priv_mode = info.mode;
266             priv->priv_arg1 = info.arg1;
267             priv->priv_arg2 = info.arg2;
268 
269             break;
270 
271         default:
272             return -EOPNOTSUPP;
273     }
274     return 0;
275 }
276 
277 /* ---------------------------------------------------------------------------
278  * create default netdev_ops struct
279  */
280 static const struct net_device_ops insane_net_device_ops = {
281     .ndo_open        = insane_open,
282     .ndo_stop        = insane_close,
283     .ndo_do_ioctl    = insane_ioctl,
284     .ndo_get_stats   = insane_get_stats,
285     .ndo_start_xmit  = insane_xmit,
286     .ndo_neigh_setup = insane_neigh_setup_dev,
287 };
288 
289 /* ---------------------------------------------------------------------------
290  * The initialization function: it is used to assign fields in the structure
291  */
292 
293 /*
294  * The __init attribute has no effect (by now) in modules; it is nonetheless
295  * good practice to declare initialization functions as such, when working
296  * in Linux kernel space.
297  */
298 void insane_init(struct net_device *dev)
299 {
300     ether_setup(dev); /* assign some of the fields as "generic ethernet" */
301     memset(netdev_priv(dev), 0, sizeof(struct insane_private));
302 
303     dev->netdev_ops = &insane_net_device_ops;
```

```
304     /*
305
306     dev->open            = insane_open;
307     dev->stop            = insane_close;
308     dev->do_ioctl        = insane_ioctl;
309     dev->get_stats       = insane_get_stats;
310     dev->hard_start_xmit = insane_xmit;
311     dev->neigh_setup     = insane_neigh_setup_dev;
312
313     */
314
315     /* Assign random MAC address */
316     random_ether_addr(dev->dev_addr);
317 }
318
319 /* ------------------------------------------------------------------------
320  * module entry points
321  */
322
323 int __init insane_init_module(void)
324 {
325     int err;
326     insane_dev = alloc_netdev(sizeof(struct insane_private),
327                               "insane", insane_init);
328
329     if (!insane_dev)
330         return -ENOMEM;
331
332     if (( err = dev_alloc_name(insane_dev, insane_dev->name) )) {
333         printk(KERN_WARNING "%s: allocate name, error %i\n",
334                insane_dev->name, err);
335         return -EIO;
336     }
337
338     if (( err = register_netdev(insane_dev) )) {
339         printk(KERN_WARNING "%s: can't register, error %i\n",
340                insane_dev->name, err);
341         return -EIO;
342     }
343
344     printk("%s: device registered\n", insane_dev->name);
345     return 0;
346 }
347
348 void __exit insane_exit_module(void)
349 {
350     unregister_netdev(insane_dev);
351     printk("%s: device unregistered\n", insane_dev->name);
352     free_netdev(insane_dev);
353 }
354
355 module_init(insane_init_module);
356 module_exit(insane_exit_module);
```

A.3 insanely.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <errno.h>
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <sys/ioctl.h>
9  #include <net/if.h>
10 #include <netinet/in.h>
11 #include <arpa/inet.h>
12
13 #include "insane.h"
```

```
14  int main(int argc, char **argv)
15  {
16      char *ifname = "insane";
17      char *prgname = argv[0];
18      char *s;
19      int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
20      struct insane_userinfo info;
21      struct ifreq req;
22
23      if (sock < 0) {
24          fprintf(stderr, "%s: socket(): %s\n", prgname, strerror(errno));
25          exit(1);
26      }
27
28      strcpy(req.ifr_name, ifname);
29
30      /* check that the insane interface exists */
31      if ( ioctl(sock, SIOCGIFFLAGS, &req) < 0 ) {
32          fprintf(stderr, "%s: %s: %s", prgname, ifname, strerror(errno));
33          if (errno == ENODEV)
34              fprintf(stderr, " (did you load the module?)");
35          putc('\n', stderr);
36          exit(1);
37      }
38
39      switch(argc) {
40          case 1: /* get information */
41              req.ifr_data = (caddr_t)&info;
42              if (ioctl(sock, SIOCINSANEGETINFO, &req)<0) {
43                  fprintf(stderr, "%s: ioctl(INSANEGETINFO): %s\n", prgname,
44                          strerror(errno));
45                  exit(1);
46              }
47              printf("slave:     %s\n", info.name);
48              printf("mode:      %s\n", info.mode == INSANE_PERCENT ? "percent" :
49                     (info.mode == INSANE_TIME ? "time" : "pass"));
50              printf("arguments: %i %i\n", info.arg1, info.arg2);
51              break;
52
53          case 3:
54          case 4:
55          case 5:
56
57              /* The first argument is the slave, then the mode, then args */
58              req.ifr_data = (caddr_t)&info;
59              strncpy(info.name, argv[1], INSANE_NAMELEN);
60              info.arg1 = info.arg2 = 0;
61              if (!strcasecmp(argv[2],"pass")) {
62                  info.mode = INSANE_PASS;
63                  if (argc != 3) {
64                      fprintf(stderr, "%s: Wrong number of arguments\n",prgname);
65                      exit(1);
66                  }
67              } else if (!strcasecmp(argv[2],"percent")) {
68                  info.mode = INSANE_PERCENT;
69                  if (argc != 4) {
70                      fprintf(stderr, "%s: Wrong number of arguments\n",prgname);
71                      exit(1);
72                  }
73                  info.arg1 = (int)strtol(argv[3], &s, 0);
74                  if (*s || info.arg1 < 0 || info.arg2 > 100) {
75                      fprintf(stderr, "%s: Wrong value \"%s\"\n",
76                              prgname, argv[3]);
77                      exit(1);
78                  }
79              } else if (!strcasecmp(argv[2],"time")) {
80                  info.mode = INSANE_TIME;
81                  if (argc != 5) {
82                      fprintf(stderr, "%s: Wrong number of arguments\n",prgname);
83                      exit(1);
84                  }
```

```
85                  info.arg1 = (int)strtol(argv[3], &s, 0);
86                  if (*s || info.arg1 < 0) {
87                      fprintf(stderr, "%s: Wrong value \"%s\"\n",
88                              prgname, argv[3]);
89                      exit(1);
90                  }
91                  info.arg2 = (int)strtol(argv[4], &s, 0);
92                  if (*s || info.arg2 < 0) {
93                      fprintf(stderr, "%s: Wrong value \"%s\"\n",
94                              prgname, argv[4]);
95                      exit(1);
96                  }
97              } else {
98                  fprintf(stderr, "%s: Wrong mode \"%s\"\n", prgname, argv[2]);
99              }
100
101             /* ok, now pass this information to the device */
102             if (ioctl(sock, SIOCINSANESETINFO, &req)<0) {
103                 fprintf(stderr, "%s: ioctl(INSANESETINFO): %s\n", prgname,
104                         strerror(errno));
105                 exit(1);
106             }
107             break;
108
109         default:
110             fprintf(stderr, "%s: Wrong number of arguments\n", prgname);
111             exit(1);
112     }
113     return 0;
114 }
```

A.4 atl1e_xmit_frame()

```
1  static int atl1e_xmit_frame(struct sk_buff *skb, struct net_device *netdev)
2  {
3      struct atl1e_adapter *adapter = netdev_priv(netdev);
4      unsigned long flags;
5      u16 tpd_req = 1;
6      struct atl1e_tpd_desc *tpd;
7
8      /* insane */
9
10     int accept; /* accept this packet or drop it */
11     static unsigned long randval;
12
13     switch (adapter->priv_mode) {
14
15         case INSANE_PERCENT:
16             if (!randval) randval = jiffies;
17             randval = ((randval * 1103515245) + 12345) & 0x7fffffff;
18             accept = (randval % 100) < adapter->priv_arg1;
19         break;
20
21         case INSANE_TIME:
22             randval = jiffies % (adapter->priv_arg1 + adapter->priv_arg2);
23             accept = randval < adapter->priv_arg1;
24         break;
25
26         case INSANE_PASS:
27             default: /* unknown mode: default to pass */
28             accept = 1;
29     }
30
31     if (!accept) {
32         kfree_skb(skb);
33         /* Mark packet as dropped */
34         adapter->hw_stats.tx_abort_col++;
35         return 0;
36     }
```

```
37      /* sane */
38
39      if (test_bit(__AT_DOWN, &adapter->flags)) {
40          dev_kfree_skb_any(skb);
41          return NETDEV_TX_OK;
42      }
43
44      if (unlikely(skb->len <= 0)) {
45          dev_kfree_skb_any(skb);
46          return NETDEV_TX_OK;
47      }
48
49      tpd_req = atl1e_cal_tdp_req(skb);
50
51      if (!spin_trylock_irqsave(&adapter->tx_lock, flags))
52          return NETDEV_TX_LOCKED;
53
54      if (atl1e_tpd_avail(adapter) < tpd_req) {
55          /* no enough descriptor, just stop queue */
56          netif_stop_queue(netdev);
57          spin_unlock_irqrestore(&adapter->tx_lock, flags);
58          return NETDEV_TX_BUSY;
59      }
60
61      tpd = atl1e_get_tpd(adapter);
62
63      if (unlikely(adapter->vlgrp && vlan_tx_tag_present(skb))) {
64          u16 vlan_tag = vlan_tx_tag_get(skb);
65          u16 atl1e_vlan_tag;
66
67          tpd->word3 |= 1 << TPD_INS_VL_TAG_SHIFT;
68          AT_VLAN_TAG_TO_TPD_TAG(vlan_tag, atl1e_vlan_tag);
69          tpd->word2 |= (atl1e_vlan_tag & TPD_VLANTAG_MASK) <<
70                  TPD_VLAN_SHIFT;
71      }
72
73      if (skb->protocol == ntohs(ETH_P_8021Q))
74          tpd->word3 |= 1 << TPD_VL_TAGGED_SHIFT;
75
76      if (skb_network_offset(skb) != ETH_HLEN)
77          tpd->word3 |= 1 << TPD_ETHTYPE_SHIFT; /* 802.3 frame */
78
79      /* do TSO and check sum */
80
81      if (atl1e_tso_csum(adapter, skb, tpd) != 0) {
82          spin_unlock_irqrestore(&adapter->tx_lock, flags);
83          dev_kfree_skb_any(skb);
84          return NETDEV_TX_OK;
85      }
86
87      atl1e_tx_map(adapter, skb, tpd);
88      atl1e_tx_queue(adapter, tpd_req, tpd);
89
90      netdev->trans_start = jiffies;
91      spin_unlock_irqrestore(&adapter->tx_lock, flags);
92      return NETDEV_TX_OK;
93  }
```

A.5 struct net_device

```
1   /*
2    *      The DEVICE structure.
3    *      Actually, this whole structure is a big mistake.  It mixes I/O
4    *      data with strictly "high-level" data, and it has to know about
5    *      almost every data structure used in the INET module.
6    *
7    *      FIXME: cleanup struct net_device such that network protocol info
8    *      moves out.
9    */
```

```
10 struct net_device
11 {
12         /*
13          * This is the first field of the "visible" part of this structure
14          * (i.e. as seen by users in the "Space.c" file).  It is the name
15          * the interface.
16          */
17         char                    name[IFNAMSIZ];
18
19         /* device name hash chain */
20         struct hlist_node       name_hlist;
21
22         /* snmp alias */
23         char                    *ifalias;
24
25         /*
26          *      I/O specific fields
27          *      FIXME: Merge these and struct ifmap into one
28          */
29         unsigned long           mem_end;        /* shared mem end       */
30         unsigned long           mem_start;      /* shared mem start     */
31         unsigned long           base_addr;      /* device I/O address   */
32         unsigned int            irq;            /* device IRQ number    */
33
34         /*
35          *      Some hardware also needs these fields, but they are not
36          *      part of the usual set specified in Space.c.
37          */
38
39         unsigned char           if_port;        /* Selectable AUI, TP,..*/
40         unsigned char           dma;            /* DMA channel          */
41
42         /* Net device state */
43         unsigned long           state;
44
45         struct list_head        dev_list;
46         struct list_head        napi_list;
47
48         /* Net device features */
49         unsigned long           features;
50
51 #define NETIF_F_SG              1       /* Scatter/gather IO. */
52 #define NETIF_F_IP_CSUM         2       /* Can checksum TCP/UDP over IPv4. */
53 #define NETIF_F_NO_CSUM         4       /* Does not require checksum. */
54 #define NETIF_F_HW_CSUM         8       /* Can checksum all the packets. */
55 #define NETIF_F_IPV6_CSUM       16      /* Can checksum TCP/UDP over IPV6 */
56 #define NETIF_F_HIGHDMA         32      /* Can DMA to high memory. */
57 #define NETIF_F_FRAGLIST        64      /* Scatter/gather IO. */
58 #define NETIF_F_HW_VLAN_TX      128     /* Transmit VLAN hw acceleration */
59 #define NETIF_F_HW_VLAN_RX      256     /* Receive VLAN hw acceleration */
60 #define NETIF_F_HW_VLAN_FILTER  512     /* Receive filtering on VLAN */
61 #define NETIF_F_VLAN_CHALLENGED 1024    /* Device cannot handle VLAN packets */
62 #define NETIF_F_GSO             2048    /* Enable software GSO. */
63 #define NETIF_F_LLTX            4096    /* LockLess TX - deprecated. Please */
64                                         /* do not use LLTX in new drivers */
65 #define NETIF_F_NETNS_LOCAL     8192    /* Does not change network namespaces */
66 #define NETIF_F_GRO             16384   /* Generic receive offload */
67 #define NETIF_F_LRO             32768   /* large receive offload */
68
69 #define NETIF_F_FCOE_CRC        (1 << 24) /* FCoE CRC32 */
70
71         /* Segmentation offload features */
72 #define NETIF_F_GSO_SHIFT       16
73 #define NETIF_F_GSO_MASK        0x00ff0000
74
75 #define NETIF_F_TSO             (SKB_GSO_TCPV4 << NETIF_F_GSO_SHIFT)
76 #define NETIF_F_UFO             (SKB_GSO_UDP << NETIF_F_GSO_SHIFT)
77 #define NETIF_F_GSO_ROBUST      (SKB_GSO_DODGY << NETIF_F_GSO_SHIFT)
78 #define NETIF_F_TSO_ECN         (SKB_GSO_TCP_ECN << NETIF_F_GSO_SHIFT)
79 #define NETIF_F_TSO6            (SKB_GSO_TCPV6 << NETIF_F_GSO_SHIFT)
80 #define NETIF_F_FSO             (SKB_GSO_FCOE << NETIF_F_GSO_SHIFT)
```

```
81          /* List of features with software fallbacks. */
82  #define NETIF_F_GSO_SOFTWARE    (NETIF_F_TSO | NETIF_F_TSO_ECN | NETIF_F_TSO6)
83  #define NETIF_F_GEN_CSUM        (NETIF_F_NO_CSUM | NETIF_F_HW_CSUM)
84  #define NETIF_F_V4_CSUM         (NETIF_F_GEN_CSUM | NETIF_F_IP_CSUM)
85  #define NETIF_F_V6_CSUM         (NETIF_F_GEN_CSUM | NETIF_F_IPV6_CSUM)
86  #define NETIF_F_ALL_CSUM        (NETIF_F_V4_CSUM | NETIF_F_V6_CSUM)
87
88          /*
89           * If one device supports one of these features, then enable them
90           * for all in netdev_increment_features.
91           */
92  #define NETIF_F_ONE_FOR_ALL     (NETIF_F_GSO_SOFTWARE | NETIF_F_GSO_ROBUST | \
93                                   NETIF_F_SG | NETIF_F_HIGHDMA |         \
94                                   NETIF_F_FRAGLIST)
95
96          /* Interface index. Unique device identifier    */
97          int                     ifindex;
98          int                     iflink;
99
100         struct net_device_stats stats;
101
102 #ifdef CONFIG_WIRELESS_EXT
103         /* List of functions to handle Wireless Extensions (instead of ioctl).
104          * See <net/iw_handler.h> for details. Jean II */
105         const struct iw_handler_def *   wireless_handlers;
106         /* Instance data managed by the core of Wireless Extensions. */
107         struct iw_public_data * wireless_data;
108 #endif
109         /* Management operations */
110         const struct net_device_ops *netdev_ops;
111         const struct ethtool_ops *ethtool_ops;
112
113         /* Hardware header description */
114         const struct header_ops *header_ops;
115
116         unsigned int            flags;  /* interface flags (a la BSD)   */
117         unsigned short          gflags;
118         unsigned short          priv_flags; /* flags invisible to userspace. */
119         unsigned short          padded; /* padding added by alloc_netdev() */
120
121         unsigned char           operstate; /* RFC2863 operstate */
122         unsigned char           link_mode; /* mapping policy to operstate */
123
124         unsigned                mtu;    /* interface MTU value          */
125         unsigned short          type;   /* interface hardware type      */
126         unsigned short          hard_header_len; /* hardware hdr length */
127
128         /* extra head- and tailroom the hardware may need, but not in all cases
129          * can this be guaranteed, especially tailroom. Some cases also use
130          * LL_MAX_HEADER instead to allocate the skb.
131          */
132         unsigned short          needed_headroom;
133         unsigned short          needed_tailroom;
134
135         struct net_device       *master; /* Pointer to master device of a group,
136                                           * which this device is member of.
137                                           */
138
139         /* Interface address info. */
140         unsigned char           perm_addr[MAX_ADDR_LEN]; /* hw address */
141         unsigned char           addr_len;       /* hardware address length */
142         unsigned short          dev_id;         /* for shared network cards */
143
144         spinlock_t              addr_list_lock;
145         struct dev_addr_list    *uc_list;       /* Secondary mac addresses */
146         int                     uc_count;       /* Number of installed ucasts */
147         int                     uc_promisc;
148         struct dev_addr_list    *mc_list;       /* Multicast mac addresses */
149         int                     mc_count;       /* Number of installed mcasts */
150         unsigned int            promiscuity;
151         unsigned int            allmulti;
```

```
152         /* Protocol specific pointers */
153 #ifdef CONFIG_NET_DSA
154         void                    *dsa_ptr;       /* dsa specific data */
155 #endif
156         void                    *atalk_ptr;     /* AppleTalk link       */
157         void                    *ip_ptr;        /* IPv4 specific data   */
158         void                    *dn_ptr;        /* DECnet specific data */
159         void                    *ip6_ptr;       /* IPv6 specific data */
160         void                    *ec_ptr;        /* Econet specific data */
161         void                    *ax25_ptr;      /* AX.25 specific data */
162         struct wireless_dev     *ieee80211_ptr; /* IEEE 802.11 specific data,
163                                                    assign before registering */
164 /*
165  * Cache line mostly used on receive path (including eth_type_trans())
166  */
167         unsigned long           last_rx;        /* Time of last Rx */
168         /* Interface address info used in eth_type_trans() */
169         unsigned char           dev_addr[MAX_ADDR_LEN];
170         unsigned char           broadcast[MAX_ADDR_LEN];
171
172         struct netdev_queue     rx_queue;
173
174         struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
175
176         /* Number of TX queues allocated at alloc_netdev_mq() time  */
177         unsigned int            num_tx_queues;
178
179         /* Number of TX queues currently active in device  */
180         unsigned int            real_num_tx_queues;
181
182         unsigned long           tx_queue_len; /* Max frames per queue allowed */
183         spinlock_t              tx_global_lock;
184 /*
185  * One part is mostly used on xmit path (device)
186  */
187         /* These may be needed for future network-power-down code. */
188         unsigned long           trans_start; /* Time (in jiffies) of last Tx */
189
190         int                     watchdog_timeo; /* used by dev_watchdog() */
191         struct timer_list       watchdog_timer;
192
193         /* Number of references to this device */
194         atomic_t                refcnt ____cacheline_aligned_in_smp;
195
196         /* delayed register/unregister */
197         struct list_head        todo_list;
198         /* device index hash chain */
199         struct hlist_node       index_hlist;
200
201         struct net_device       *link_watch_next;
202
203         /* register/unregister state machine */
204         enum { NETREG_UNINITIALIZED=0,
205                NETREG_REGISTERED,       /* completed register_netdevice */
206                NETREG_UNREGISTERING,    /* called unregister_netdevice */
207                NETREG_UNREGISTERED,     /* completed unregister todo */
208                NETREG_RELEASED,         /* called free_netdev */
209                NETREG_DUMMY,            /* dummy device for NAPI poll */
210         } reg_state;
211
212         /* Called from unregister, can be used to call free_netdev */
213         void (*destructor)(struct net_device *dev);
214
215 #ifdef CONFIG_NETPOLL
216         struct netpoll_info     *npinfo;
217 #endif
218
219 #ifdef CONFIG_NET_NS
220         /* Network namespace this network device is inside */
221         struct net              *nd_net;
222 #endif
```

```
223         /* mid-layer private */
224         void                    *ml_priv;
225
226         /* bridge stuff */
227         struct net_bridge_port  *br_port;
228         /* macvlan */
229         struct macvlan_port     *macvlan_port;
230         /* GARP */
231         struct garp_port        *garp_port;
232         /* class/net/name entry */
233         struct device           dev;
234         /* space for optional statistics and wireless sysfs groups */
235         struct attribute_group  *sysfs_groups[3];
236
237         /* rtnetlink link ops */
238         const struct rtnl_link_ops *rtnl_link_ops;
239
240         /* VLAN feature mask */
241         unsigned long vlan_features;
242
243         /* for setting kernel sock attribute on TCP connection setup */
244 #define GSO_MAX_SIZE            65536
245         unsigned int            gso_max_size;
246
247 #ifdef CONFIG_DCB
248         /* Data Center Bridging netlink ops */
249         struct dcbnl_rtnl_ops *dcbnl_ops;
250 #endif
251
252 #if defined(CONFIG_FCOE) || defined(CONFIG_FCOE_MODULE)
253         /* max exchange id for FCoE LRO by ddp */
254         unsigned int            fcoe_ddp_xid;
255 #endif
256
257 #ifdef CONFIG_COMPAT_NET_DEV_OPS
258         struct {
259                 int (*init)(struct net_device *dev);
260                 void (*uninit)(struct net_device *dev);
261                 int (*open)(struct net_device *dev);
262                 int (*stop)(struct net_device *dev);
263                 int (*hard_start_xmit) (struct sk_buff *skb,
264                                         struct net_device *dev);
265                 u16 (*select_queue)(struct net_device *dev,
266                                     struct sk_buff *skb);
267                 void (*change_rx_flags)(struct net_device *dev,
268                                         int flags);
269                 void (*set_rx_mode)(struct net_device *dev);
270                 void (*set_multicast_list)(struct net_device *dev);
271                 int (*set_mac_address)(struct net_device *dev,
272                                        void *addr);
273                 int (*validate_addr)(struct net_device *dev);
274                 int (*do_ioctl)(struct net_device *dev,
275                                 struct ifreq *ifr, int cmd);
276                 int (*set_config)(struct net_device *dev, struct ifmap *map);
277                 int (*change_mtu)(struct net_device *dev, int new_mtu);
278                 int (*neigh_setup)(struct net_device *dev,
279                                    struct neigh_parms *);
280                 void (*tx_timeout) (struct net_device *dev);
281                 struct net_device_stats* (*get_stats)(struct net_device *dev);
282                 void (*vlan_rx_register)(struct net_device *dev,
283                                          struct vlan_group *grp);
284                 void (*vlan_rx_add_vid)(struct net_device *dev,
285                                         unsigned short vid);
286                 void (*vlan_rx_kill_vid)(struct net_device *dev,
287                                          unsigned short vid);
288 #ifdef CONFIG_NET_POLL_CONTROLLER
289                 void (*poll_controller)(struct net_device *dev);
290 #endif
291         };
292 #endif
293 };
```

Bibliografia

[1] P Baran, "On distributed communication networks", *IEEE Trans. On Commun. Systems*, , no. 12, 1964.

[2] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection", *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[3] James Martin, Kathleen Kavanagh Chapman, and CORPORATE Inc. The Arben Group, *SNA: IBM's networking solution*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[4] S. Wecker, "Dna: The digital network architecture", April 1980.

[5] R. Bush and D. Meyer, "Some Internet Architectural Guidelines and Philosophy", RFC 3439 (Informational), Dec. 2002.

[6] G. Malkin and R. Minnear, "RIPng for IPv6", RFC 2080 (Proposed Standard), Jan. 1997.

[7] J. Moy, "OSPF Version 2", RFC 2328 (Standard), Apr. 1998.

[8] D. Oran, "OSI IS-IS Intra-domain Routing Protocol", RFC 1142 (Informational), Feb. 1990.

[9] J. Postel, "Transmission Control Protocol", RFC 793 (Standard), Sept. 1981, Updated by RFC 3168.

[10] Jonathan Stone and Craig Partridge, "When the crc and tcp checksum disagree", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.7611, 2000.

[11] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control", RFC 2581 (Proposed Standard), Apr. 1999, Updated by RFC 3390.

[12] J. Postel, "User Datagram Protocol", RFC 768 (Standard), Aug. 1980.

[13] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340 (Proposed Standard), Mar. 2006.

[14] U. Demir and O. Aktas, "Raptor versus reed solomon forward error correction codes", 2006.

[15] Gregor N. Purdy, *Linux Iptables*, O'Reilly, 08 2004.

[16] S. Hemminger, "Network emulation with netem", April 2005.

[17] Inc. Free Software Foundation, "Status of c99 features in gcc", http://gcc.gnu.org/c99status.html, March 2009.

[18] J. Liedtke, "On micro-kernel construction", 1995.

[19] Alessandro Rubini, "Virtual network interfaces", http://www.linux.it/ rubini/docs/vinter/, April 2000.

[20] Brian W. Kernighan, Dennis Ritchie, and Dennis M. Ritchie, *C Programming Language (2nd Edition)*, Prentice Hall PTR, March 1988.

[21] J. Postel, "Internet Control Message Protocol", RFC 792 (Standard), Sept. 1981, Updated by RFCs 950, 4884.

[22] The Iperf team, "Iperf, a tool for measuring tcp and udp performance", http://iperf.sourceforge.net/, March 2003.