POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Prototipazione di un acceleratore di traffico per dispositivi embedded basato su eBPF/XDP

Politecnico di Torino

Relatori
prof. Fulvio Risso
ing. Federico Parola

Candidato
Niccolò Giraudo

Dicembre 2022

Abstract

Velocizzare l'inoltro dei pacchetti sulle reti di calcolatori rappresenta una delle sfide più importanti del routing moderno. L'accelerazione del traffico è oggetto di una ricerca continua, che ha portato alla nascita di dispositivi di rete di ultima generazione che integrano componenti hardware in grado di gestire i pacchetti, migliorando le prestazioni e risparmiando capacità di processing della CPU per altri task. Questa proposta presenta però degli svantaggi legati ai costi economici dei chip fisici e ad una minore flessibilità rispetto ad altri approcci. Ultimamente, per ovviare a questi problemi, sono nate numerose soluzioni per l'accelerazione del traffico a livello software, tra le quali spicca la tecnologia eBPF. Lo svolgimento di questo lavoro si basa sullo sfruttamento di questa tecnologia per accelerare la gestione del traffico di rete elaborato da un router software-driven. Verrà proposta una soluzione modulare e le sue prestazioni verranno confrontate con altri approcci. Come dimostrato dai risultati, le performance ottenute dalla soluzione proposta sono migliori di quelle ottenibili tramite le funzionalità messe a disposizione dal kernel Linux, pur mantenendo un alto livello di integrazione con i tradizionali meccanismi di configurazione di rete.

Ringraziamenti

L'intero mio percorso universitario è stato lungo e travagliato, ma non mi è mai mancato il sostegno delle persone che mi circondano e mi regalano continuamente momenti di respiro dagli affanni che mi creo io stesso.
Sfrutterò la tesi per ringraziare chi in questi anni di università mi ha aiutato a continuarla.
In primis voglio ringraziare la mia famiglia, i miei genitori che mi hanno dato fiducia anche quando tutto non filava nel verso giusto. I miei fratelli, Jacopo e Carola: squadra indivisibile e sostegno morale l'un dell'altro.
I miei amici ronchesi, sinonimo di libertà e gioia: coloro che se c'è da fare festa non si tirano indietro nemmeno se li prendi a colpi di cannone.
Tutta la famiglia del CuneoPedona rugby, ora Cuneo Saluzzo, che mi ha regalato negli anni grandi amicizie, legami profondi e momenti di sollievo e mi ha insegnato la voglia di lottare senza mai tirarsi indietro.
I Corinzi, compagni di vita, laboratori, lezioni, girate. Colleghi senza i quali non avrei saputo passare nemmeno un quarto degli esami dati. Grazie, di cuore.
Tutti gli amici che qui non posso citare altrimenti si aggiungerebbero 20 pagine a questa tesi già lunga di suo: sappiate che qualunque persona che io abbia conosciuto, mi ha regalato energie per arrivare fino a qui.
In ultimo, ma non per importanza, Muci (un pezzo pane) che mi è stata vicino, negli alti e nei bassi, ricordandomi periodicamente di cosa io fossi in grado di fare: se sono qui, alla fine di questo percorso universitario, lo devo a lei.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Motivazione

L'obiettivo che la tesi svolta presso Tiesse si propone di perseguire è quello di verificare le funzionalità di una macchina che utilizzi la tecnologia eBPF/XDP per accelerare il traffico di rete. Ad oggi l'azienda si è sempre basata su una delle seguenti opzioni: una macchina "bare metal" che sfrutta il kernel di Linux per ridirigere il traffico, un ottimizzatore di Netflow e un acceleratore hardware. Quest'ultima soluzione è quella più utilizzata nel mondo dei costruttori di router, in quanto permette prestazioni che nessun'altra proposta è in grado di raggiungere. Si preferisce questo approccio perché in grado di adattarsi automaticamente alle politiche di gestione del traffico, in quanto utilizza degli algoritmi che sfruttano l'Intelligenza Artificiale. A questa proposta è però necessario associare un costo economico dovuto all'integrazione dei chip fisici all'interno dei dispositivi: l'idea di poter utilizzare acceleratori software origina da questa esigenza.
Le tecnologie viste in precedenza però presentano grandi limitazioni dal punto di vista prestazionale. L'introduzione di eBPF/XDP consentirebbe di aumentare notevolmente le performance garantendo un'alta flessibilità al sistema.

1.2 Tiesse S.p.A.

Tiesse è una società privata che produce router internet, apparecchiature di rete e dispositivi M2M/IoT. Ha sede a Ivrea, nel nord del Piemonte, in Italia.
Ha altre tre sedi in tutta Italia: una a Torino, una ad Avezzano (vicino a L'Aquila) e uno a Roma.
Uno dei più gradi punti di forza di Tiesse è sicuramente il completo controllo della filiera, dalla progettazione del prodotto alla produzione e la filiera di distribuzione.
I prodotti Tiesse sono orientati ad un approccio Zero-Touch Provisioning, che permette al cliente di installare i dispositivi senza la necessità che qualcuno lo configuri in maniera corretta.
In costante collaborazione con il Politecnico di Torino, Tiesse propone un reparto di ricerca e sviluppo sempre in movimento. Questo porta allo sviluppo di tecnologie.
I prodotti Tiesse sono particolarmente orientati a soluzioni Enterprise e Business.

Capitolo 2

Background

2.1 eBPF

eBPF (Extended Berkeley Packet Filter) è una tecnologia che permette di iniettare codice nel kernel di Linux ed eseguirlo in una sandbox. eBPF è l'evoluzione di BPF, un packet filter nato a fine 1992 che, seguendo un paradigma *store&hold*, permetteva l'inserimento di numerosi filtri all'interno del sistema operativo. Due dei problemi principali di BPF erano la mancanza di informazioni sullo stato e il basso grado di ottimizzazione che era implementato all'interno di questa tecnologia: infatti, il costo computazionale di più filtri in parallelo risultava lineare al numero di filtri.
A causa di queste problematiche si è iniziato a cercare una soluzione alternativa e più performante: dal kernel 4.4 si è iniziato ad utilizzare eBPF. Questo nuovo modo di intendere il packet filter è diventato popolare non solo in ambito networking ma in tutto il mondo della programmazione lato kernel.

2.1.1 Code injection

Una delle peculiarità di eBPF è quella del "code injection": permette di iniettare codice runtime nel kernel di Linux. Questo aspetto semplifica notevolmente la programmazione a basso livello. Rende infatti possibile la creazione di programmi

senza l'aggiunta di moduli ulteriori all'interno del kernel o la modifica del codice sorgente di quest'ultimo.
Il bytecode generato è eseguito in una virtualCPU che permette di ricevere ed elaborare del codice in modo sicuro.
L'obbiettivo diviene la creazione di programmi il più specializzati possibili e, di conseguenza, minimali. Questo porta ad un risultato che aspira ad ottenere un codice più preciso possibile che però possa cambiare dinamicamente.

2.1.2 Safety

La tecnologia eBPF presenta obiettivi simili al suo predecessore BPF (o cBPF) riguardo alla sicurezza. Sono limitazioni molto stringenti dal punto di vista del programmatore ma sono necessarie: scrivendo codice kernel senza alcuna protezione si rischia di danneggiare il sistema in esecuzione in caso di errore. Questa motivazione ha portato all'introduzione di barriere dal punto di vista della programmazione. Solo un bytecode verificato e corretto può essere eseguito dal sistema. Colui che si occupa di questo aspetto è il verifier, un componente software che è adibito al controllo del codice.
Un'ulteriore limitazione introdotta da BPF ed eBPF risulta essere l'imposizione di un numero di istruzioni finito: questa caratteristica presenta degli ostacoli intrinsechi tra i quali si presentano l'assenza di cicli infiniti ed un numero massimo di istruzioni eseguibili (pari a 4096). Per aggirare queste problematiche si è rimossa la possibilità di scrivere programmi che contenessero dei loop. Solo nelle ultime versioni è stata introdotta l'opportunità di scrivere loop predictable, ovvero con un numero di cicli definito a priori.
Un'ultima limitazione che è stata resa necessaria da eBPF risulta essere un consumo di memoria definito: in questo caso è sufficiente rimuovere la possibilità di allocare dinamicamente delle variabili.

2.1.3 Verifier

Il verifier è il componente software che si occupa del controllo del codice. Viene eseguito dopo aver compilato il programma e ha il compito di verificare che tutte le limitazioni elencate nel punto precedente siano rispettate. [1] Il controllo viene effettuato in due fasi distinte:

1. Nella prima il verifier si occupa di esaminare il DAG risultante dal flusso di comandi fornito; più precisamente si occupa di appurare che non ci siano delle istruzioni irraggiungibili dal programma.

2. La seconda fase invece analizza l'intero codice simulando ogni possibile percorso dell'algoritmo. In questa fase si osservano gli stati dei registri e dello stack, in modo da prevenire eventuali accessi in zone di memoria alle quali non è previsto che si acceda.

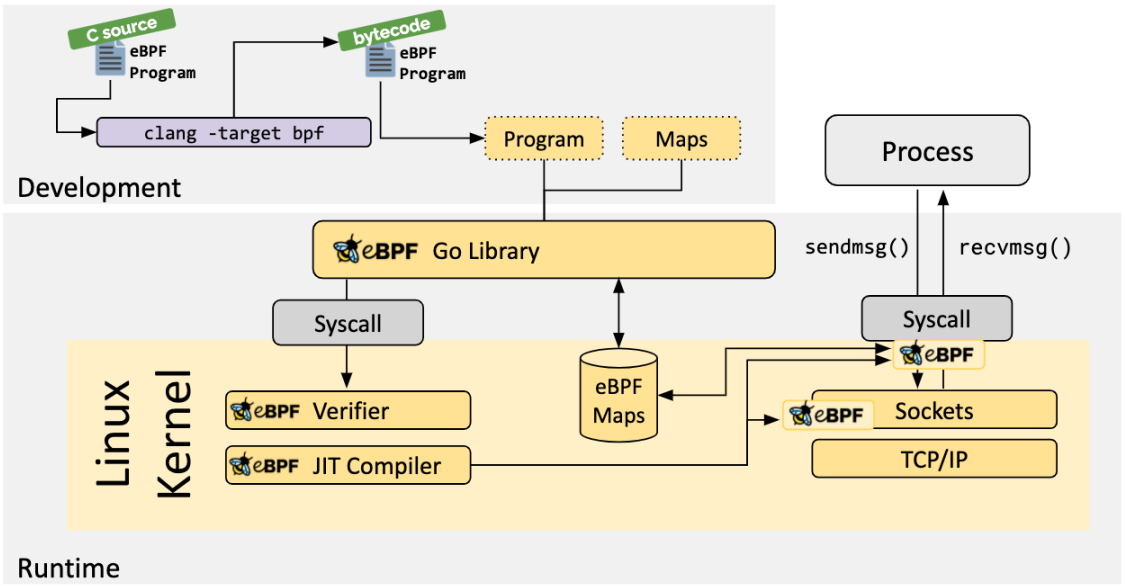

Figura 2.1. Schema del funzionamento del verifier

2.1.4 vCPU

Per garantire la sicurezza del sistema, eBPF viene eseguito in una sandbox, ovvero un meccanismo per eseguire un programma in uno spazio limitato.
La virtual CPU di cui è fornito eBPF è incaricato della scrittura e della trasformazione del codice C in codice Assembly adatto per l'infrastruttura eBPF. Infatti questo processore è strutturato in modo che possa emulare l'hardware necessario ad ogni istruzione.
In questo modo il sistema garantisce flessibilità al sistema senza creare problemi di sicurezza, in quanto il codice è eseguito da un interprete.

2.1.5 Hook-point multipli

La tecnologia eBPF garantisce una moltitudine di hookpoint, ovvero dei luoghi virtuali all'interno del sistema operativo in cui poter caricare il programma eBPF a seconda della necessità del programmatore. Sono siti a diversi livelli del kernel di Linux per garantire una maggiore eterogeneità.

I principali hookpoint sono:

- XDP
- TC-ingress
- TC-egress
- eBPF/socket
- AF_XDP

Dalla figura 2.2 si può intuire come, all'interno dello stack di rete di Linux, siano presenti diverse tipologie di hookpoint diffusi in tutto il sistema. Come si evincerà nella sezione (2.1.10), quella indicata in figura è la posizione in presenza di un

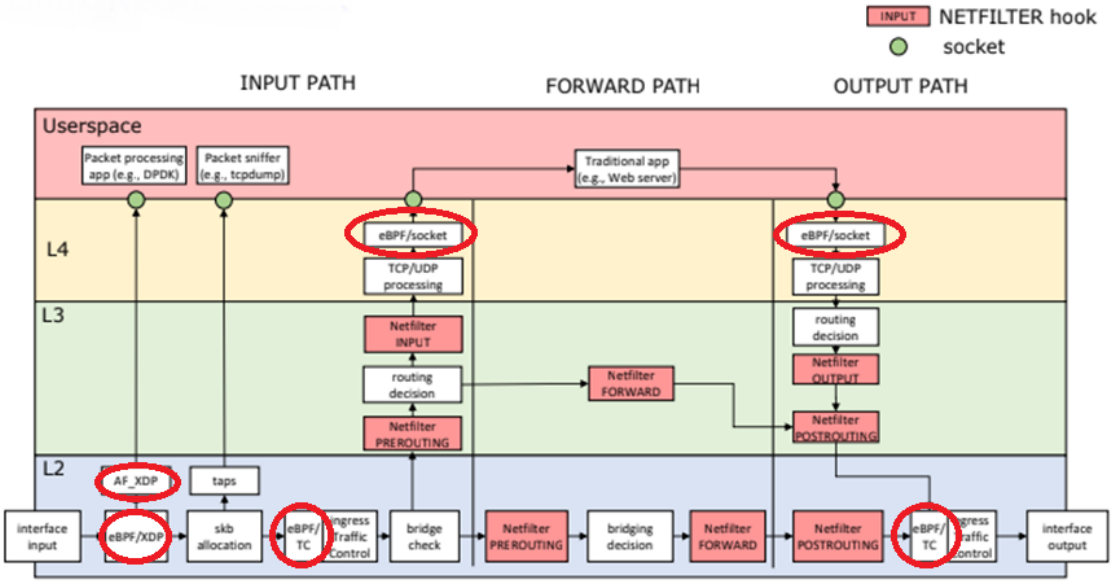

Figura 2.2. Hookpoint eBPF all'interno nello stack di rete di Linux

modulo installato in modalità "xdp_driver", ma ne esistono altre che si posizionano prima o dopo a quella mostrata.

2.1.6 Mappe

L'utilizzo delle mappe nasce dall'esigenza del programmatore di mantenere lo stato del programma eBPF in qualche zona di memoria (un esempio possono essere i contatori di pacchetti gestiti, non assimilabili nella memoria di pacchetto). Altre motivazioni sono identificabili nella necessità di accedere ai dati memorizzati dallo userspace o doverli inviare in kernel space [2]. Un'altra causa che ha contribuito all'utilizzo delle mappe, è l'esigenza di avere un metodo per poter condividere dati tra programmi eBPF diversi, proposta che può tornare utile come vedremo nella sezione 2.1.9.

Di conseguenza, sono state definite le mappe, ovvero zone di memoria formattate: è quindi necessario indicare il tipo di mappa desiderata. I principali tipi sono:

- Hash
- Array
- LongestPrefixMatch
- CGROUP_STORAGE

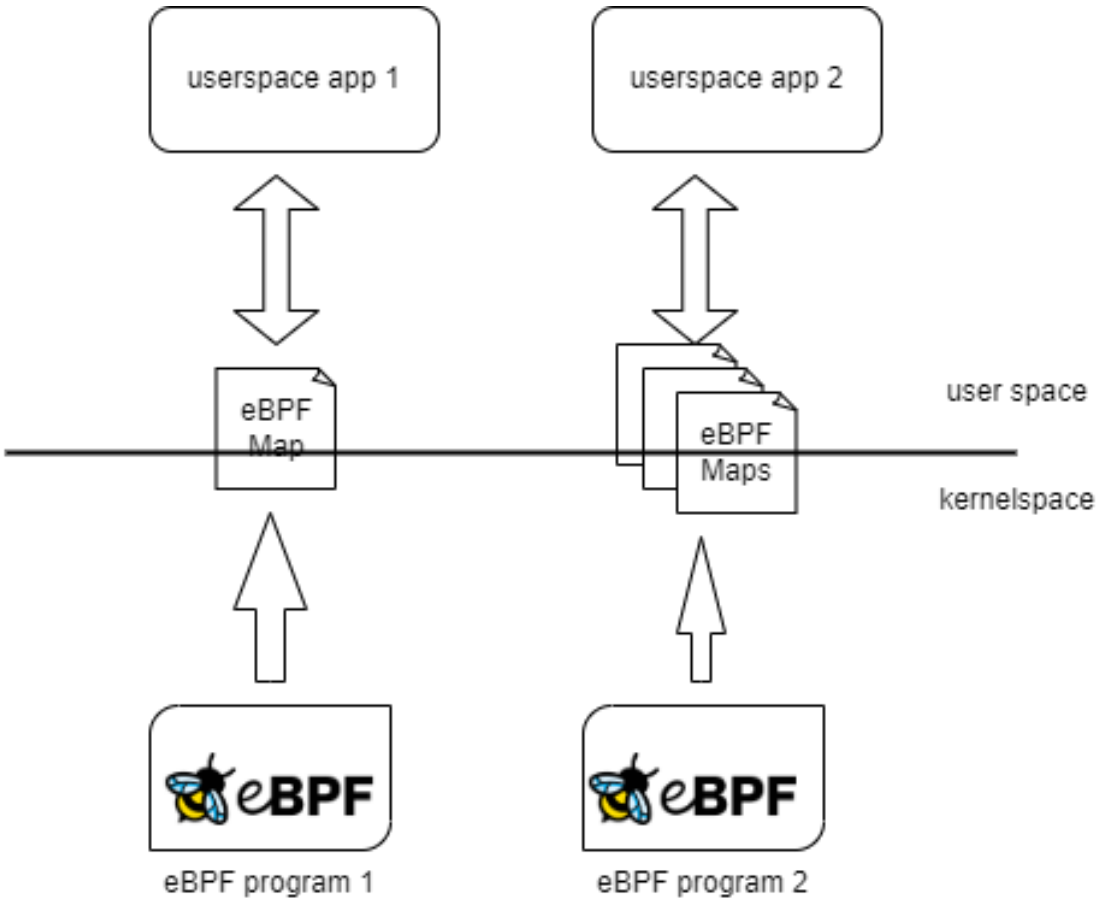

Figura 2.3. Approccio zero-copy delle mappe eBPF

Per quanto riguarda i primi due tipi specificati, è opportuno specificare che esistono anche le versioni LRU e PER_CPU. Le prime forniscono alla mappa la semantica LRU (Least Recently Used) per la gestione dell'eliminazione dei dati dalla tabella. La modalità PER_CPU permette di legare la mappa ad un singolo

core in esecuzione in modo da non avere dei costi di sincronizzazione tra i diversi core creando delle zone di memoria distinte.
Per spostare i dati in userspace, esistono diversi metodi: eBPF fornisce la modalità zero-copy per la gestione delle memorie. Questa pratica consiste nel creare una memoria condivisa tra kernel e user space, in modo da non avere il costo della copia dei dati da una memoria all'altra a scapito della sicurezza dei dati, in quanto possono essere modificati da entrambe le componenti collegate.

2.1.7 Helpers

Gli helper sono funzioni che facilitano l'interazione con diversi programmi eBPF. Non sono presenti nel BPF classico (anche chiamato cBPF). Possono essere usati per manipolare le mappe o scambiare informazioni con il kernel del sistema operativo in modo del tutto sicuro per quest'ultimo. Considerando che esistono diversi tipi di programmi eBPF, ognuno eseguito in contesti diversi, ogni tipo di programma può chiamare solo un sottoinsieme di questi helper. Internamente, i programmi eBPF chiamano il codice già compilato di tale funzione evitando richieste esterne e senza provocare alcun costo ulteriore in termini di prestazioni [3].

2.1.8 Portability

Il bytecode BPF è indipendente dall'hardware, quindi può essere eseguito su qualunque architettura.

2.1.9 Tail-calls

La tecnologia eBPF consente la creazione di catene di programmi che interagiscono tra di loro, permettendo ad esempio ad un programma di eBPF di chiamarne un

altro. L'esecuzione del secondo processo non ritorna al precedente, quindi non è vista come una chiamata ricorsiva che incrementa lo stack del sistema, bensì lo riusa.

Questa funzionalità può essere usata per costruire infrastrutture più complesse eludendo il numero di istruzioni massimo che eBPF impone. Per passare valori da un programma all'altro è possibile utilizzare una mappa condivisa a cui due o più programmi hanno accesso.

Tuttavia, è importante sottolineare una limitazione: le tail-call sono possibili solo tra programmi dello stesso tipo.

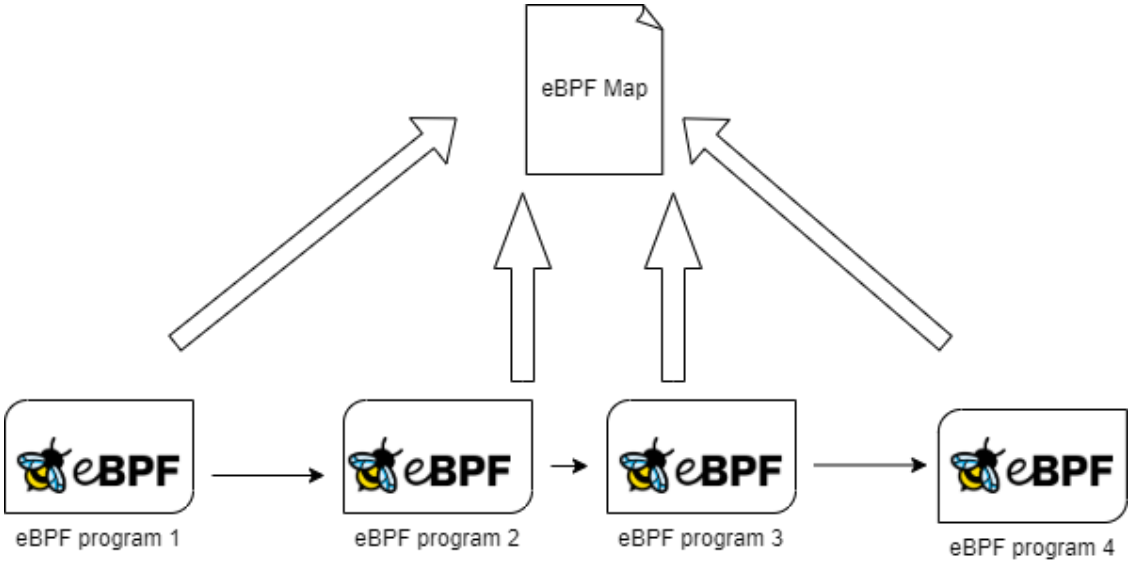

Figura 2.4. Condivisione dei dati con una mappa in una tail-call eBPF

2.1.10 eBPF/XDP

L'hookpoint su cui si basa tutto il lavoro di tesi che seguirà è quello in XDP (eXpress Data Path) [4]. XDP è un hookpoint ad alte performance su cui si può inserire un programma eBPF. È in grado di gestire un solo programma alla volta, per cui non è possibile caricare più programmi contemporaneamente come è permesso invece in

TC.
È un hookpoint che funziona solamente in ingresso, non è possibile inserirlo in egress. Questo ha causato delle limitazioni durante questo lavoro di tesi.
Esistono tre tipi differenti di XDP, in base al sistema utilizzato.

- XDP_GENERIC: questa modalità di caricamento permette di emulare il reale funzionamento di XDP, in quando avviene dopo l'allocazione dell'skb, quindi già all'interno dello stack di rete di Linux. Questa tipologia di hookpoint è utilizzata principalmente in fase di test e non in produzione.

- XDP_NATIVE: anche chiamato xdp_driver, è posizionato direttamente all'interno del driver della scheda di rete, del quale richiede il supporto per poter funzionare. È utilizzato in produzione in quanto permette di avere delle prestazioni nettamente migliori rispetto a xdp_generic, come si può vedere nella figura 2.5.

- XDP_OFFLOAD: potendo spostarsi ancora più "indietro" nel percorso del pacchetto all'interno dell'apparato preso in considerazione, troviamo la funzionalità di xdp_offload. Questa modalità di caricamento del programma permette di eseguire tutta la gestione del pacchetto, interamente sulla scheda di rete, richiedendo però un supporto adeguato da parte di quest'ultima e dell'interfaccia.

L'hookpoint eBPF, riceve una struttura di tipo XDP_MD e come valore di ritorno può assumere quattro diversi valori che indicano quattro operazioni diverse da effettuare [6]:

- XDP_DROP: questa operazione permette di scartare il pacchetto in ingresso. È utilizzata molto come funzione perché permette di effettuare l'*early drop packet*, ovvero scartare il pacchetto prima che entri nello stack di rete e quindi nel sistema operativo del dispositivo. Si usa principalmente per mitigare attacch DDoS.

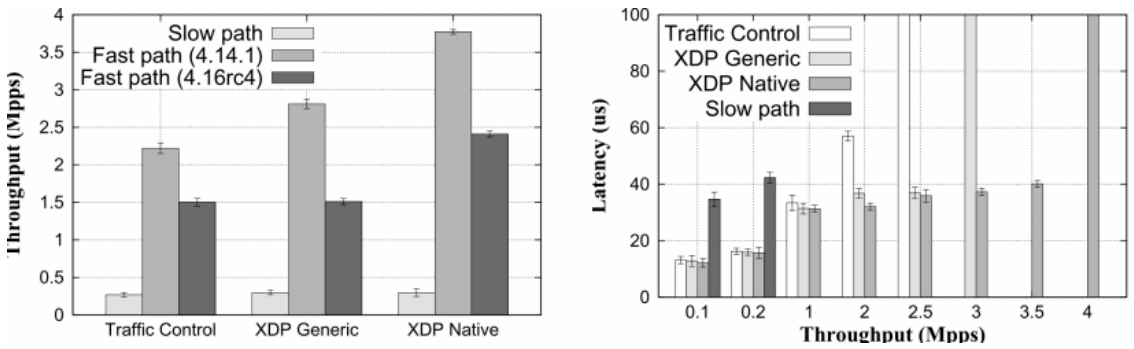

Figura 2.5. Confronto tra XDP_NATIVE e XDP_GENERIC tratta dall'articolo [5]

- XDP_PASS: Questa operazione permette di passare il pacchetto ricevuto al kernel, ovvero facendogli proseguire il percorso nello stack di rete.
- XDP_REDIRECT: Permette di bypassare lo stack di rete e reindirizzare il pacchetto ad un'altra NIC o sulla rete.
- XDP_TX: Invia il pacchetto ricevuto (che potrebbe essere stato modificato) sulla stessa interfaccia sul quale è stato ricevuto
- XDP_ABORTED: questa operazione scarta il pacchetto in seguito ad un errore del programma.

2.2 Stato dell'arte

Da parte di Tiesse era stato proposto l'utilizzo di FlowOffload, un ottimizzatore di netfilter, per velocizzare la trasmissione del traffico.

2.2.1 Routing in Linux

La figura 2.2 mostra, in modo estremamente sintetico, i principali passaggi che un pacchetto deve eseguire all'interno dello stack di rete di Linux prima di poter essere inoltrato verso un altro dispositivo.

Il pacchetto, intercettato dal driver della scheda di rete, viene passato al sistema operativo, risvegliato tramite un interrupt. In questo istante, è necessario creare la struttura skb, ovvero una struttura socket buffer, una struttura molto complessa ma considerata una delle più importanti della programmazione di rete [7] perché qualsiasi pacchetto inviato o ricevuto è gestito formattato in questa maniera.
Una volta terminata questa fase, si prosegue nella sezione indicata come "Link Layer". In questa porzione di sistema operativo, si esegue un cosiddetto "bridge check": se il responso è positivo il pacchetto viene gestito interamente come una trama di livello 2 della pila ISO-OSI. Si attiva quindi una fase di PREROUTING gestita da Netfilter, quindi si prendono la decisione riguardo quale interfaccia di uscita usare per inoltrare la trama passando quindi ad una fase di POSTROUTING.
Se il "bridge check" è negativo si inoltra ai livelli superiori ed il pacchetto viene trattato come un pacchetto di livello 3 della pila ISO-OSI. Dualmente a quanto detto prima, vengono eseguite una fase di PREROUTING e POSTROUTING gestite da Netfilter, intervallate da una di decisione nella quale si interroga la tabella di routing.
I costi principali di questa soluzione sono l'allocazione della struttura skb ed il numero di decisioni da prendere per ogni pacchetto.

2.2.2 Acceleratori Hardware

Gli acceleratori hardware sono dispositivi in grado di ottimizzare la gestione dei pacchetti. Si sfrutta l'intelligenza artificiale per poter determinare i percorsi per il traffico.
L'avvento di chip programmabili ha permesso di poter utilizzare dei programmi più flessibili. Per queste soluzioni, infatti, si utilizzano dei circuiti programmabili (FPGA) o dei circuiti integrati (ASIC) in modo da poter garantire le prestazioni migliori. I primi permettono all'utente di configurare direttamente i parametri

elettrici. Gli ASIC, invece, risultano essere creati appositamente per risolvere un'applicazione di calcolo.

2.2.3 xdp_router

È già presente un router implementato in eBPF: questo include un programma in XDP che gestisce completamente la trasmissione dei pacchetti. Questo progetto, da cui si è ispirato questo lavoro di tesi per migliorarlo e renderlo più completo, si basa sull'utilizzo dell'helper BPF_FIB_LOOKUP(), il quale permette di accedere alle tabelle di routing dell'intero sistema. Questo approccio però, è meno prestante della soluzione costruita in collaborazione con Tiesse.

2.2.4 FlowOffload

La funzione di flow offloading fornita dal sistema operativo Linux consiste nel popolare una tabella (flowtable) inserendo delle regole per implementare un fastpath nel flusso di dati.
Un pacchetto che trova una voce corrispondente nella tabella di flusso viene trasmesso all'interfaccia di uscita tramite *neigh ()*, quindi ignorando il classico percorso di inoltro IP. Nel caso in cui non vi sia alcuna voce corrispondente nella tabella di flusso, il pacchetto segue il percorso di inoltro IP classico.
La flowtable utilizzata risulta essere una tabella di tipo hash che utilizza una chiave più complessa di quella scelta nella soluzione attuata sfruttando la tecnologia eBPF. Oltre agli indirizzi IP e le porte (destinazione e sorgente) ed il protocollo di livello 3, la chiave include anche il protocollo di incapsulamento a livello 2 della pila ISO-OSI e l'interfaccia di ingresso [24].
Il programma incaricato di popolare la flowtable accelerando il traffico TCP e UDP è quello rappresentato nella porzione di codice 2.2.4. Quello che si evidenzia, è la

semplicità con cui è possibile modificare la tabella di flusso, ovvero eseguendo un semplice programma di tipo Bash.

```
#!/sbin/nft -f
flush ruleset

table inet x {
        flowtable f {
                hook ingress priority 0; devices = { eth1, eth2 };
        }
        chain y {
                type filter hook forward priority 0; policy accept
  ;
                ip protocol tcp flow offload @f
                ip protocol udp flow offload @f
                counter packets 0 bytes 0

                ct state established,related counter accept
                ip protocol tcp accept
                ip protocol udp accept
        }
}
```

Listing 2.1. Programma incaricato di riempire la flowtable

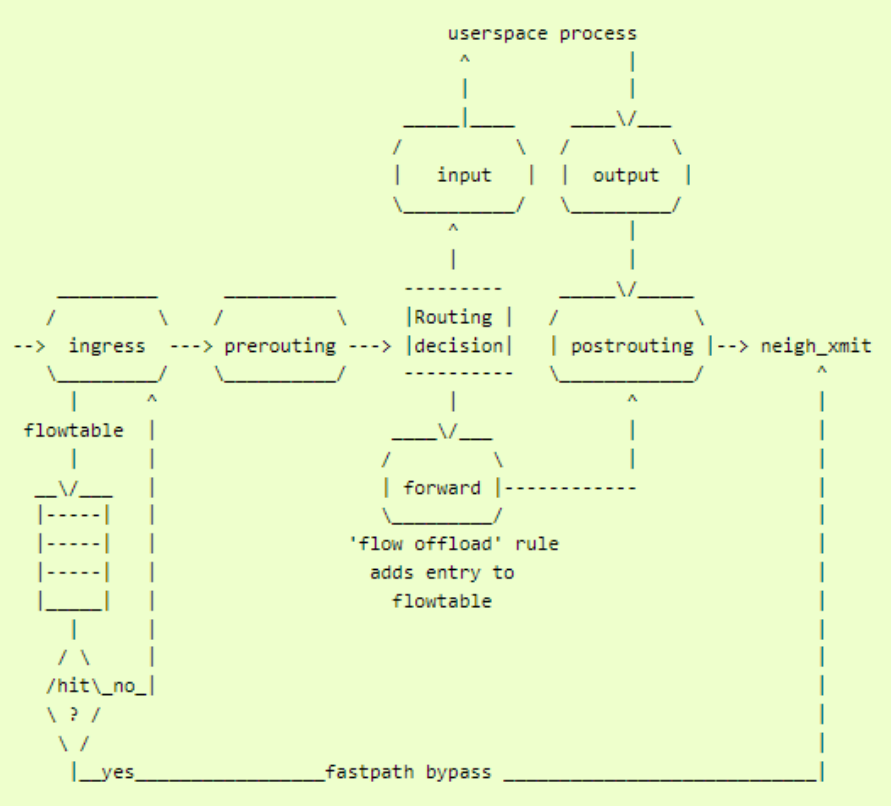

Figura 2.6. Hook di netfilter e le interazioni con FlowOffload

Capitolo 3

Architettura del prototipo

Per emulare il comportamento degli acceleratori di rete già presenti in Tiesse, si è scelto di mettere a punto un prototipo che seguisse un approccio di backlearning al problema. Si è preferito questo metodo di sviluppo per essere certi di intercettare qualsiasi operazione che debba essere fatta sul pacchetto.

3.1 Schema generale del prototipo

Il principio seguito per la creazione di questa infrastruttura è quella dell'implementazione del backlearning delle regole di routing e delle policy adibite al traffico. Più nello specifico, si tratta di utilizzare un approccio reattivo alle modifiche all'interno del sistema.
Infatti, il primo pacchetto di una sessione, riconoscibile tramite un apposito identificativo, attraverserà il kernel senza nessun intervento da parte dell'acceleratore eBPF. Questo primo pacchetto passerà nello stack di rete di Linux in modo che il sistema operativo lo possa gestire: i parametri di uscita saranno salvati in una mappa, un'apposita zona di memoria. Così facendo, solamente i primi dati ricevuti appartenenti ad una specifica sessione saranno elaborati dal kernel, quelli successivi saranno gestiti completamente dall'acceleratore software basato su eBPF, più precisamente saranno inoltrati direttamente dal programma posizionato all'ingresso.

Per mantenere però allineati il kernel e l'infrastruttura, è necessario aggiornare periodicamente le variabili memorizzate nella mappa. Per questo si utilizza un ulteriore programma in userspace in modo da poter accedere alle entry e poterle eliminare se ritenute troppo datate. Durante la gestione del primo pacchetto di una determinata sessione, verrà infatti associato anche un campo *age* così da poter valutare da quanto è stata creata la chiave presa in esame ed eventualmente procedere alla sua cancellazione. In userspace, sarà poi compito di un ulteriore programma dover esaminare periodicamente tutte le entry memorizzate nella mappa. Si è scelta questa soluzione in quanto ritenuta la più semplice e meno costosa dal punto di vista prestazionale.

Per permettere questo tipo di sistema si è reso necessario l'introduzione di più programmi eBPF: uno in ingresso sull'hookpoint XDP, uno in uscita su quello TC-egress e uno in userspace.

Infatti, come già visto nella sezione 2.1.10, l'hookpoint eBPF/XDP, quello incaricato dell'accelerazione, permette solamente il caricamento del programma sull'interfaccia di ingresso. Questo implica necessariamente il dover cercare un'alternativa: la soluzione la si trova inserendo un secondo programma nell'hookpoint TC-egress di eBPF, in grado di intercettare il pacchetto in uscita salvandolo in una mappa di tipo hash denominata *xdp_map*, pinnata all'interno del file system virtuale bpffs. Questo permette così una fruizione rapida e semplice dei dati. I flussi di pacchetti si possono distinguere tramite un ID di una sessione: questo identificativo è composto da cinque componenti distinte, parti di una singola struttura denominata *session_id*. Questi campi formano una quintupla che è costituita da: IP sorgente, IP destinazione, porta sorgente, porta destinazione e numero di protocollo a livello 3 della pila ISO-OSI utilizzato. Il programma TC-egress permette di salvare il pacchetto in uscita nella mappa condivisa. Una volta completata quest'azione, il tutto verrà inoltrato all'interfaccia di uscita continuando così il proprio percorso all'interno dello stack di rete di Linux per permettere di essere inviato all'esterno.

Il programma in XDP infatti, se non trova l'identificativo del pacchetto nella mappa adibita al routing, per fare in modo che venga reindirizzato, dovrà procedere con il classico passaggio attraverso il kernel, tragitto che avrebbe dovuto seguire nel caso in cui l'acceleratore non fosse stato presente. In caso contrario, si utilizzerà il metodo XDP_REDIRECT tramite l'helper BPF_REDIRECT che sarà in grado di inviare all'interfaccia recuperata dalla mappa.

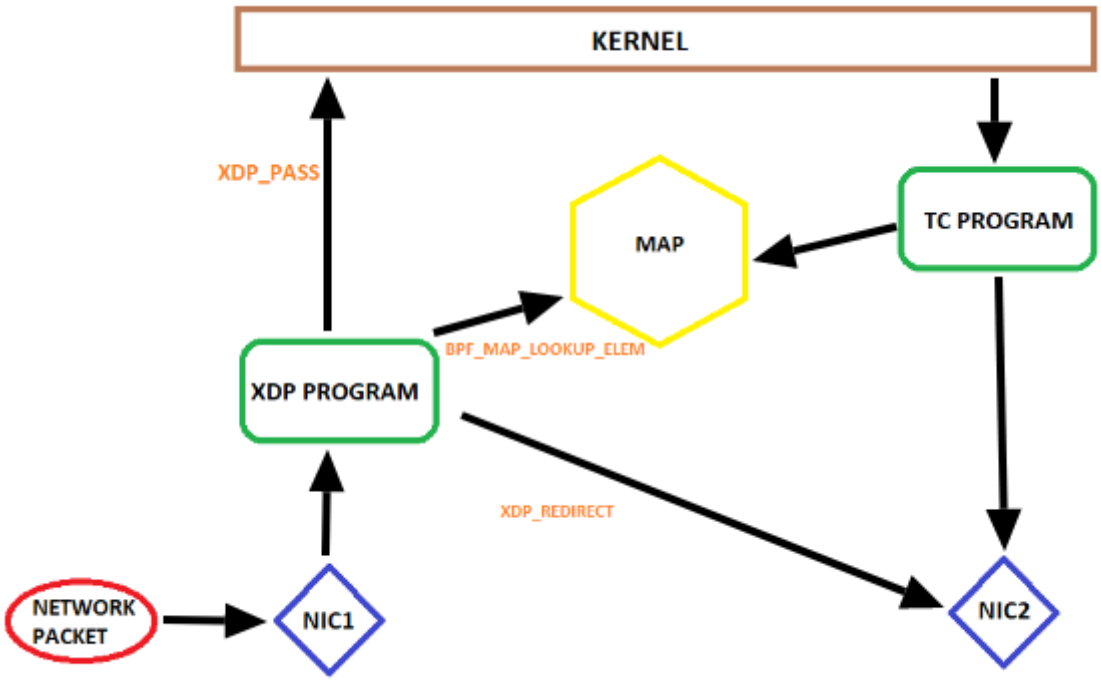

Figura 3.1. Schema del flusso di dati a livello kernel

Dalla figura (3.1) si può avere una panoramica dell'architettura del sistema creato. Come precedentemente menzionato, l'intero sistema applica un processo di backlearning per stabilire le interfacce di uscita su cui ridirigere i singoli pacchetti. Di conseguenza, solo il primo di una sessione che arriva al programma XDP verrà inviato al kernel, altrimenti verrà inoltrato alla scheda di rete corretta accelerando così la trasmissione dei dati.

3.1.1 Gestione della concorrenza

La concorrenza tra programmi differenti è stato un aspetto caratterizzante di questo programma utilizzandone più di uno in contemporanea.
Questa problematica è stata elusa, almeno nella prima fase, con l'utilizzo degli helper che, essendo operazioni atomiche, garantivano l'assenza di corse critiche. Applicando un paradigma produttore-consumatore, si veniva a creare un sistema in cui il primo, ovvero colui che scriveva nella mappa (programma in TC-egress), opera in totale sicurezza, così come il consumatore (programma in XDP che agiva in sola lettura).
Con l'introduzione delle statistiche (contatori di pacchetti e bytes totali trasmessi) la questione si è complicata ulteriormente in quanto non è stato più possibile utilizzare gli helpers come soluzione. È risultato necessario sfruttare la funzione atomica fornite dal kernel *___synch_fetch_and_add()* per poter eseguire un aggiornamento senza incorrere in più operazioni che vengono eseguite contemporaneamente.
Un'altra soluzione testata è stata quella dell'impiego delle mappe PER_CPU per la gestione delle statistiche. Questo tipo di mappa, come già visto nella sezione 2.1.6, evita l'incorrere in corse critiche.

3.2 LS1046A Freeway Board

La scheda LS1046A-FRWY prevede dei meccanismi di accodamento e gestione del frame reordering automatico e si basa sull'utilizzo di QMan, un Queue Manager che gestisce i movimenti dei dati attraverso programmi software e componenti hardware [10]. La distribuzione Ubuntu Main, utilizzata sulla scheda Freeway, utilizza un servizio di systemd chiamato "fmc.service" che, se inattivo, distribuisce i flussi di dati su più processori diversi.

3.2.1 QMan: schedulazione e riordino dei pacchetti

Questo tipo di sistema è inserito per permettere l'utilizzo di tutte le CPU in contemporanea. Se non si sfruttasse, un singolo flusso di dati sarebbe gestito da un processore, rischiando così un sovraccarico di lavoro su tale core. Questo tipo di approccio, più classico ed utilizzato, è sfruttato per evitare il fenomeno dei pacchetti *Out-Of-Order*, ovvero l'arrivo dei pacchetti alle interfacce di uscita in ordine casuale dovuto al fatto che ogni processore ha una propria coda interna di istruzioni da eseguire provocando così dei disallineamenti temporali tra le CPU. Per sfruttare tutti i core disponibili in contemporanea per l'elaborazione dei dati del medesimo flusso risulta quindi necessario dover utilizzare un gestore dei pacchetti che metta in relazione i frame in arrivo con i processori che dovranno gestirli.
Avere un componente hardware che rio+rdina i dati in uscita, limiterebbe così le alterazioni dovute alla problematica dell'Out-Of-Order che si verrebbe a creare.

Default Scheduling

Il *Default Scheduling*, implementato dalla scheda, consiste nell'utilizzo di una coda di frame (Frame Queue, FQ) del livello 2 della pila ISO-OSI, le quali sono attive fino a quando utilizzano il proprio credito o si svuotano. Una volta che l'FQ è stata liberata, QMan può riattivarla ed assegnarla ad un core, non necessariamente quello che la aveva in gestione precedentemente [11].
Questo crea una "sticky affinity", ovvero una relazione tra l'FQ e la CPU che gestisce la coda. Aumentare il credito di un FQ, permetterebbe un'affinità più stretta, ma a scapito della granularità dei pacchetti.

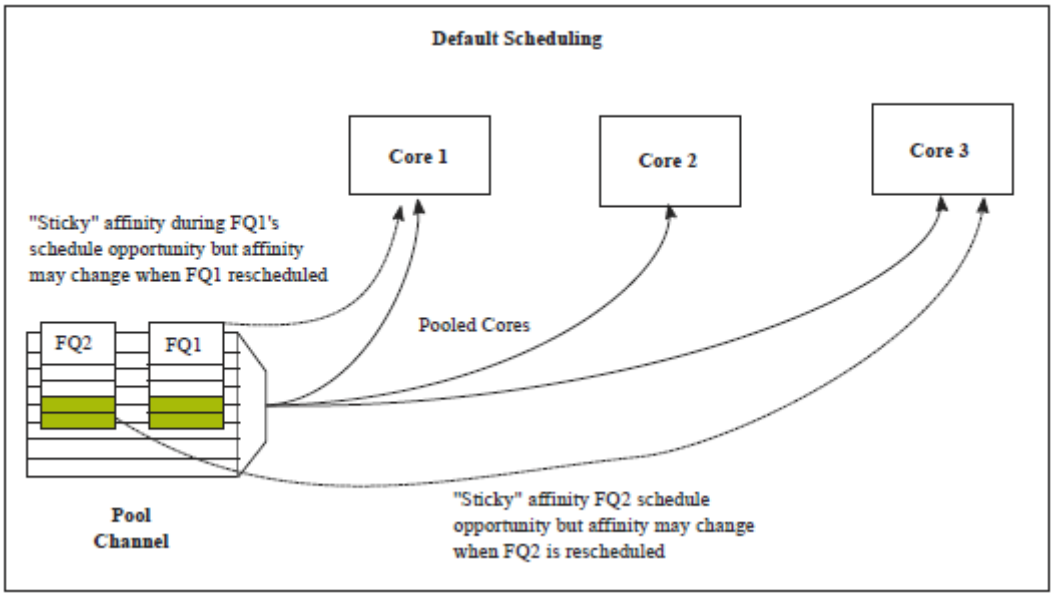

Figura 3.2. Gestione della sticky affinity da parte di QMan

QMan: Order Definition/ Restoration

Con l'utilizzo della "sticky affinity" si è risolto solamente il problema dell'utilizzo dei processori in contemporanea per analizzare lo stesso flusso di dati. Quello che manca a questo approccio per renderlo una soluzione funzionante, è la garanzia che all'uscita, i pacchetti saranno mantenuti nell'ordine in cui sono arrivati al router. Questo si può garantire solamente tenendo traccia dei pacchetti in ingresso: infatti QMan permette di applicare un tag di 14 bit ad ogni frame incrementandolo di volta in volta. Questo permette di identificare univocamente i dati che arrivano al router. In uscita, un punto di ripristino dell'ordine ritarda il posizionamento di un frame sull'FQ fino a quando non viene raggiunto il numero di sequenza successivo previsto.

Ogni FQ che partecipa al riordinamento, ha un proprio NESN (Next Expected Sequence Number). Esiste inoltre una soglia configurabile oltre la quale non si segnala al sistema di non aspettare più i pacchetti precedenti, ma inoltrare tutto

agli FQ di uscita. Nella circostanza in cui il frame mancante arrivasse, si può configurare la scheda in modo che lo rifiuti o accodi in ritardo. [12]

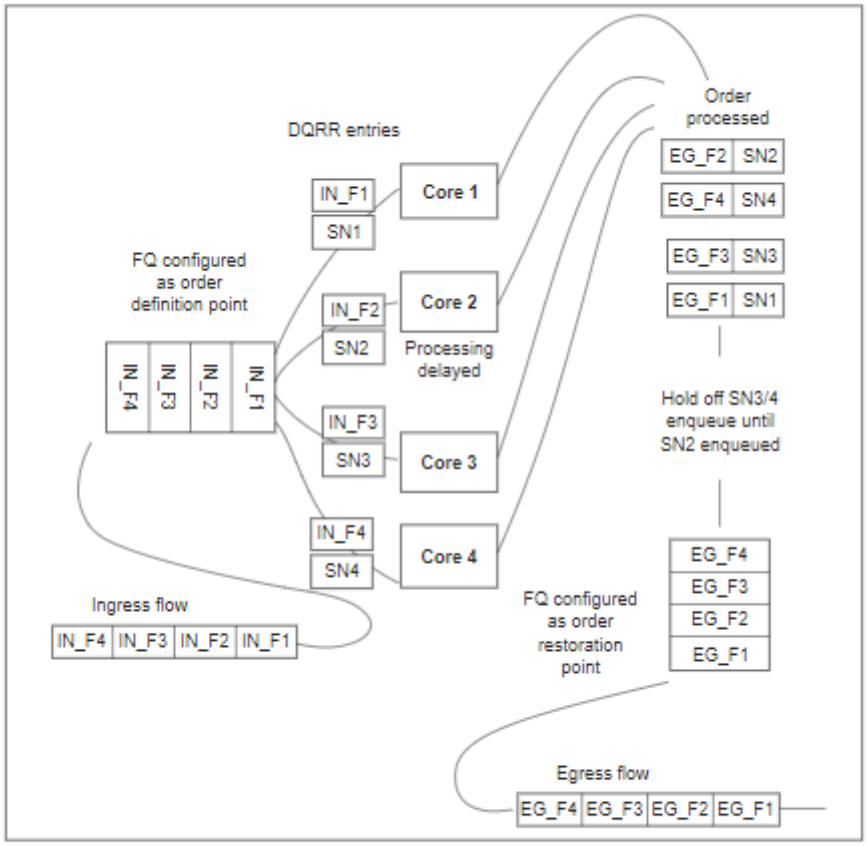

Figura 3.3. Gestione del reordering da parte di QMan

Capitolo 4

Implementazione del prototipo

4.1 libbpf

libbpf [13] è una libreria di supporto per eBPF e XDP che facilita l'uso della tecnologia eBPF. Al suo interno contiene degli helper aggiuntivi che facilitano la programmazione da parte dell'utente.
Include sia funzioni in kernel space che in userspace e semplifica la gestione delle mappe sfruttando la definizione BTF.

4.2 iproute2

Iproute2 è una suite di istruzioni a linea di comando. Permette di agire a qualsiasi livello dello stack di rete di Linux, andando così a sostituire le varie collezioni di tool già esistenti come *ifconfig*, *route* o *netstat*: al suo interno si possono infatti trovare i comandi *ip*, *arpd*, *tc* o *ss* indispensabili per accedere alle risorse di rete da linea di comando. Questo tool contiene infatti delle funzioni adibite alla gestione di programmi e strutture eBPF in grado di aiutare il loading dei programmi o il controllo dei flussi di dati.
Una distribuzione di iproute2 risultava già presente all'interno del kernel costruito

con Buildroot, ma questo non era in grado di gestire la definizione delle mappe in BTF-style e allo stesso tempo non era capace di poter gestire correttamente la libreria libbpf. Per questo si è scelto di installare sulla macchina la versione 5.17.0 disponibile sul repository del kernel di Linux [14]. Si è scelta quella distribuzione per via della sua data di rilascio (molto recente), ma allo stesso tempo si è rilevata essere molto stabile relativamente al nostro scopo.
Contenuta in questa libreria, si può trovare un file header che include la ridefinizione di tutti gli header BPF, il file bpf.h[15]. Questo documento definisce inoltre i valori dei flag necessari al sistema eBPF per allocare variabili o definire tipi.
Stabilisce inoltre un prototipo della struttura ricevuta dal programma XDP che verrà poi arricchita però dalla libreria libbpf con ulteriori campi necessari al forwarding dei pacchetti.

```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
struct xdp_md {
  __u32 data;
  __u32 data_end;
};
```

Listing 4.1. Struttura xdp_md inserita in iproute2

Per caricare un file BPF compilato nel kernel è sufficiente utilizzare questo comando in modalità super-user:

```
ip link set *interface_name* xdp obj *file object*
```

4.3 Acceleratore XDP

L'intero sistema, come già visto nella figura 3.1, utilizza un meccanismo di backlearning che permette di applicare le regole ai pacchetti solo una volta che il primo di tale sessione viene intercettato dal dispositivo, in modo che quelli successivi siano

reindirizzati senza passare tramite il kernel Linux. Il sistema, come già detto, è formato da tre programmi principali: i primi due sono adibiti alla gestione del routing e dell'aggiornamento delle statistiche, il terzo invece è addetto alla pulizia e alla stampa dei dati nella mappa. In seguito, si discuterà nello specifico come ogni programma esegue questi compiti e come interagiscono tra di loro.
Per quanto riguarda il routing, la mappa utilizzata per lo scambio di dati è una mappa di tipo BPF_MAP_TYPE_HASH, ovvero di tipo Hash, denominata *xdp_map*. Ogni entry di questa zona di memoria è formata da una coppia chiave/valore. In questo progetto, la chiave utilizzata è stata una struttura rinominata *session_id* definita come segue:

```
typedef struct _key{
    __be32 saddr;
    __be32 daddr;
    __be16 sport;
    __be16 dport;
    __u8 proto;
} session_id;
```

Listing 4.2. Struct session_id

La struttura che contiene i valori da esaminare invece è più complessa e consta di tre parti:

- La prima sezione è formata dai campi del pacchetto da analizzare per applicare delle modifiche nel caso in cui venga richiesto. Si affronteranno le motivazioni nella sezione dedicata al programma XDP (4.3.2).
- La seconda contiene i campi necessari per eseguire il forwarding del pacchetto. Si possono distinguere l'interfaccia di uscita e i MAC da inserire nel frame a livello 2 della pila ISO-OSI per permettere l'inoltro dei dati.
- La terza sezione rappresenta delle variabili che saranno necessarie per la programmazione: in particolare si può trovare un campo AGE che indica il tempo

espresso in nanosecondi dalla creazione della entry ed un campo *LOCK che descrive un semaforo interno a BPF per garantire l'assenza di corse critiche.

Di seguito si mostra la definizione della struttura value dedicata al routing.

```
typedef struct fields{
    __u8 tos;
    __be16 h_vlan_TCI;

    __u32 ifindex;
    unsigned char   h_dest[ETH_ALEN];
    unsigned char   h_source[ETH_ALEN];

    __u64 age;
    struct bpf_spin_lock *lock;
} _value;
```

Listing 4.3. Struct _value

4.3.1 Programma TC-egress - file tc2.c

Il primo programma analizzato è quello posizionato sul TC-egress. Si è scelto questo perché è il primo programma che il pacchetto in uscita incontra. I dati vengono ricevuti sotto forma di una struttura di tipo STRUCT SKBUFF*. Il pacchetto viene analizzato a partire dalla correttezza dei dati: infatti, se è troppo piccolo per contenere delle strutture dati adeguate, il pacchetto sarà scartato perché non conforme. Ciò potrebbe essere causato da un errore di trasmissione a livello fisico.
La struttura sarà poi esaminata al livello 2 della pila ISO-OSI, vale a dire quella equivalente al livello di Collegamento. Si tenta di fare il cast dei dati ricevuti in una struttura adatta all'ispezione dei singoli campi, ovvero una struttura di tipo STRUCT ETHHDR* che rappresenta logicamente un frame Ethernet; in caso di errore il frame verrà scartato. Lo scopo di questa valutazione è quello di ottenere una struttura adeguata per la mappa, ovvero un session_id. I dati correttamente

formattati vengono esaminati partendo dal tipo di protocollo di livello 3 della pila ISO-OSI utilizzato dal pacchetto. Gli unici protocolli accettabili per accelerare il traffico sono quello UDP e quello TCP; nel caso in cui sia uno diverso dai due appena citati, il pacchetto sarà inoltrato senza essere salvato sulla mappa.
Procedendo ad analizzare i dati, si possono ricavare così gli indirizzi IP e le porte sorgente e destinazione, ottenuti dalla STRUCT IPHDR* da inserire nella struttura *KEY creata per contenere i dati della chiave. Entrambi i protocolli sfruttano questi quattro campi.
Una volta creata la chiave, si prosegue inserendo i valori della sezione "value": utilizzando la funzione *___builtin_memcpy* si copiano gli indirizzi MAC nella struttura valore.Questo passaggio è dovuto al fatto che essi sono trattati come vettori di numeri interi, pertanto è necessario copiare il contenuto della cella di memoria. Invece, per l'interfaccia di uscita del pacchetto, si tratta di una semplice assegnazione.
Un'ultima operazione da compiere è quella del salvataggio del tempo di creazione della entry nella mappa. All'interno del sistema eBPF, non si possono sfruttare le funzioni della programmazione classica del C (pertanto chiamate a funzioni, come ad esempio KGETTIME(), non sono ammesse). eBPF fornisce però l'helper BPF_KTIME_GET_NS(), che è in grado di ritornare al chiamante il numero di nanosecondi passati dall'avvio del sistema. Richiamando questa procedura, è possibile aggiornare il campo age della struttura _value: questo permetterà di avere il tempo di creazione della entry e del suo inserimento nella mappa.
Una volta conclusa questa procedura, la coppia chiave/valore è inserita in memoria e può essere consultata dal programma XDP. Per inserirla si sfrutta l'helper BPF_MAP_UPDATE_ELEM(struct bpf_map *map, const void *key, const void *value, u64 flags) che accetta un campo "flags". Quest'ultimo, può assumere tre diversi valori:

- BPF_NOEXIST: esegue il comando solo se la chiave non esiste ancora nella mappa, creandola

- BPF_EXIST: esegue il comando solo se la chiave è già presente nella mappa, aggiornando il campo value
- BPF_ANY: esegue il comando in qualsiasi caso

L'ultima operazione che compie il programma posizionato in TC-egress è l'aggiornamento delle statistiche: esso non è uniforme e varia in base alla soluzione adottata. Una spiegazione più approfondita si può trovare nella sezione 4.3.4.

4.3.2 Programma XDP - file "router.c"

Il secondo programma preposto all'accelerazione del traffico all'interno del router è quello posizionato sull'hookpoint eBPF/XDP in ingresso. Si tratta della principale componente del sistema ed è adibito allo smistamento dei pacchetti. Il programma riceve una struttura di tipo STRUCT XDP_MD*, già descritta in precedenza (4.2) ma arricchita con la definizione interna alla libreria libbpf. In questo caso i dati ricevuti sono visualizzati utilizzando la seguente definizione:

```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
struct xdp_md {
  __u32 data;
  __u32 data_end;
  __u32 data_meta;
  /* Below access go through struct xdp_rxq_info */
  __u32 ingress_ifindex; /* rxq->dev->ifindex */
  __u32 rx_queue_index;  /* rxq->queue_index  */
  __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

Listing 4.4. Struttura xdp_md arricchita da libbpf

Per quanto concerne la corretta formattazione dei dati, il funzionamento si può definire simile a quello del programma posizionato in TC-egress. La prima azione che

viene compiuta è la verifica sulla dimensione dei dati. Infatti, come nel programma in TC-egress, la struttura xdp_md deve essere grande abbastanza da poter contenere una struttura di tipo STRUCT ETHHDR*. Nel caso in cui il pacchetto non fosse conforme, esso verrà scartato ritornando XDP_DROP.

In questo programma prima di analizzare il pacchetto e constatare a quale protocollo appartiene, bisogna verificare la presenza di eventuali tag VLAN che incapsulano il tutto. Se presente, si rimuove la struttura dati che lo circonda:

```
1  if (ether_proto == ETH_P_8021Q || ether_proto == ETH_P_8021AD) {
2          // tagged pkt on non-trunked port, drop
3          vhdr = l3hdr;
4          if (l3hdr + sizeof(struct vlan_hdr) > data_end) return
       XDP_DROP;
5
6          l3hdr += sizeof(struct vlan_hdr);
7          ether_proto = vhdr->inner_ether_proto;
8          _val.h_vlan_TCI = vhdr->vlan_id;
9      }else{
10         _val.h_vlan_TCI = -1;
11     }
```

Listing 4.5. Rimozione VLAN

Come emerge dal codice 4.3.2, la struttura l3hdr contiene i dati del pacchetto a livello 3 della pila ISO-OSI. Per rimuovere la porzione dedicata alla VLAN, si sposta in avanti l'offset di questa struttura di un valore pari a quello ritornato dalla funzione *sizeof(struct vlan_hdr);*.

Una volta esclusa questa parte, si procede a verificare il tipo di protocollo utilizzato: anche in questo caso si esaminano solo pacchetti che sfruttano il TCP oppure l'UDP. Lo scopo di questa operazione è la medesima del programma TC-egress: creare una struttura session_id che consenta di identificare la sessione a cui appartiene questo tipo di traffico. Conseguemente, si esamineranno gli indirizzi IP sorgente e destinazione e le rispettive porte. Questo passaggio permette di creare

la struttura chiave adatta per cercare il flusso all'interno della mappa xdp_map.

```
ret = bpf_map_lookup_elem(&xdp_map, &_key);

    if(ret != NULL){
      rc = S_SUCCESS;
  }else
      rc = S_FAILED;
    switch(rc) {
        case S_SUCCESS:

         _decr_ttl(ether_proto, l3hdr);
            if(ret->tos != _val.tos)
                if(ip != NULL)
                  ip->tos = ret->tos;
            if(ret->h_vlan_TCI != _val.h_vlan_TCI)
            if(vhdr != NULL)
                vhdr->vlan_id = ret->h_vlan_TCI;

            __builtin_memcpy(eth->h_source, ret->h_source, ETH_ALEN
   );
            __builtin_memcpy(eth->h_dest, ret->h_dest, ETH_ALEN);
            return bpf_redirect(ret->ifindex, 0);
            break;
        case S_FAILED:
        default:
            return XDP_PASS;
            break;
    }
```

Listing 4.6. Codice dell'inoltro

Una volta terminata questa porzione di codice, si prosegue con l'inoltro del pacchetto. Sfruttando l'helper BPF_MAP_LOOKUP_ELEM(), passandogli come

parametro la mappa in cui cercare e la chiave desiderata, si può verificare la presenza di tale entry. Infatti questa funzione ritorna il valore associato alla struct key* esaminata oppure può essere NULL se non è stato trovato. Nell'eventualità che il valore di ritorno sia nullo, il pacchetto viene trasmesso al kernel perché potrebbe trattarsi di un protocollo non preso in carica dal sistema di accelerazione. Nel caso in cui, al contrario, la ricerca abbia avuto esito positivo, i dati esaminati e presi in carica per il forwarding, viene decrementato il Time-To-Live del pacchetto, ovvero il massimo numero di passaggi che può effettuare attraverso una rete IP[16].
Una volta portata a termine questa operazione, si può procedere con la verifica dei campi inseriti nella mappa xdp_map. Ciò è dovuto al fatto che, in casi di flussi particolari, è necessario eseguire il mangling del pacchetto: in sostanza è necessario apporre delle modifiche ad alcune porzioni di dati per permettere un corretto inoltro. I principali valori da determinare sono il DSCP (DiffServ Code Points) e il tag delle LAN virtuale. Queste informazioni sono reperibili dalla mappa: infatti come illustrato nella porzione di codice 4.3, sono presenti i campi desiderati. Un semplice controllo seguito da un'eventuale assegnazione, permettono la modifica di questi valori.
Successivamente, si prosegue con l'inoltro effettivo del pacchetto. Si lavora ad un livello ISO-OSI più basso rispetto alla sezione precedente; infatti si vanno ad inserire nella struttura xdp_md ricevuta come argomento i MAC necessari per permettere di stabilire il prossimo dispositivo da raggiungere. Infine, con l'ausilio dell'helper BPF_REDIRECT(), specificando l'interfaccia di uscita (anch'essa recuperata dalla mappa), si invia il pacchetto al successivo apparato di retel, decretato dalla tabella di routing del kernel.
Dalla figura 4.1 si può visualizzare uno schema generale e riassuntivo di quello che compie il programma posizionato sull'hookpoint eBPF/XDP.

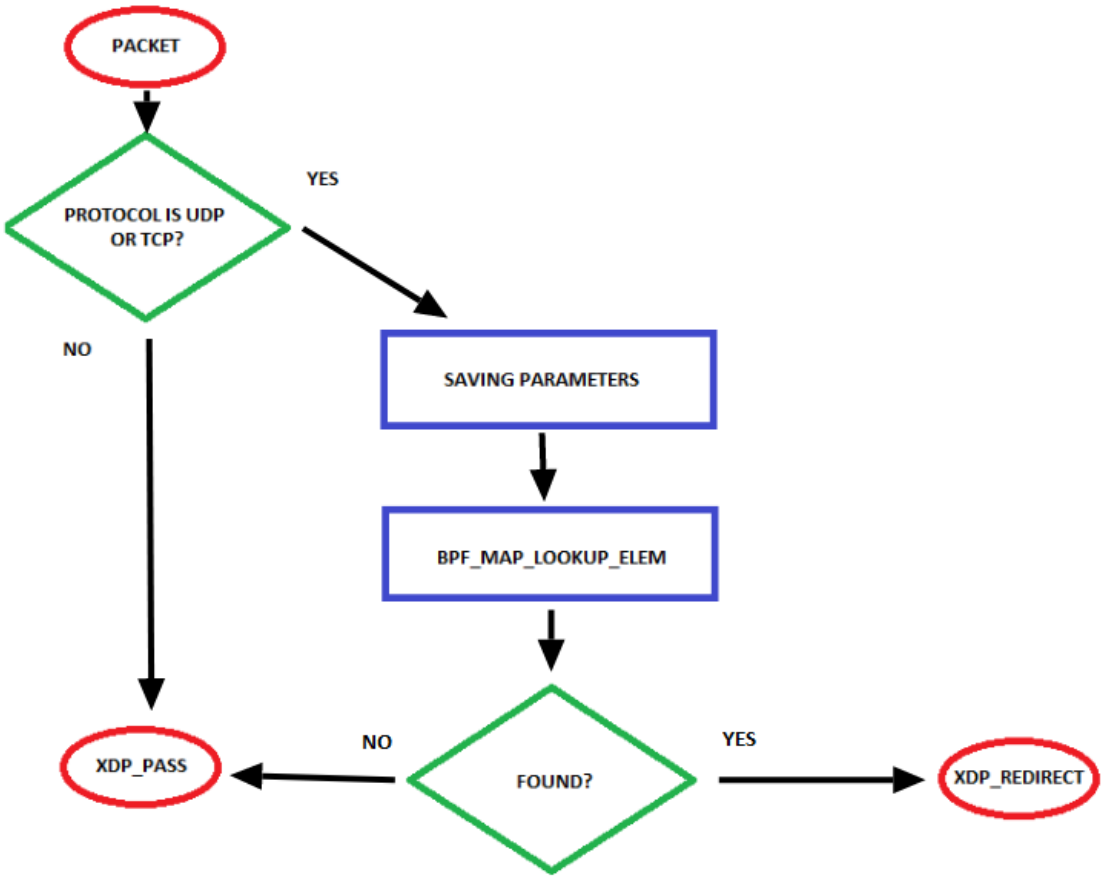

Figura 4.1. Schema del percorso dei dati all'interno del programma in XDP

4.3.3 Programma in userspace - file "entry-cleaner.c"

Il traffico gestito con i due programmi che cooperano in questa maniera per la lettura e la scrittura è funzionale all'accelerazione del traffico. Un difetto di questo sistema è la mancata gestione delle entry nelle tabelle una volta create. Questo implica due problemi principali:

- Il primo problema è sicuramente quello dell'aggiornamento dei campi. Infatti, nel caso una sessione dovesse chiudersi o dovesse cambiare dei parametri (ad esempio la variazione del campo DSCP o della VLAN, ma anche campi come l'interfaccia di uscita), il sistema risulterebbe statico, cioè non reattivo. Non vi è la possibilità di gestire cambiamenti di questo tipo o intercettare i segnali

che arrivano dal kernel utilizzando questo approccio.

- Il secondo problema di questo sistema è la presenza di un numero massimo di entry che può reggere la mappa. Al momento della creazione è necessario passare come parametro la dimensione massima desiderata. Per avere una possibilità di gestione del traffico notevole, occorreva stabilire un compromesso tra il numero massimo di entry e il costo di elaborazione nella ricerca di una determinata chiave ad ogni frame analizzato. Si è scelta una dimensione 50.000.000 entry, definite dalla costante MAX_SIZE nel file header "constants.h".

Come discusso in precedenza, creare programmi eBPF ha delle limitazioni in termini di libertà espressiva. Nella sezione 2.1.2 è emerso come sia complicato scrivere codici complessi usando questa tecnologia. Di fatto risultava quasi impensabile andare a verificare eventuali modifiche nelle tabelle del sistema del kernel ad ogni pacchetto arrivato: si è pertanto reso necessario utilizzare un altro approccio per non sovraccaricare il processore.
La soluzione che si è scelto di adottare è stata l'inserimento del campo AGE contenuto all'interno della struttura _value rappresentata in 4.3.
La soluzione è giunta gestendo la mappa e le sue modifiche in userspace, ovvero potendo accedervi in un ambiente libero da limitazioni.
Questo programma, rinominato entry-cleaner.c, è incaricato di eliminare le entry all'interno della mappa xdp_map dopo un certo tempo dalla loro creazione. Tale funzionamento permetterebbe un approccio reattivo ai problemi sopracitati (4.3.3) garantendo un continuo ricambio delle informazioni scritte nella mappa (nel caso di modifiche delle tabelle di routing o dei valori del DSCP/ToS e del tag VLAN) o rimozione (nel caso la sessione si chiuda). Si può notare come l'utilizzo di un programma in userspace sarebbe in grado di aggirare le limitazioni dovute alla creazione di loop e quindi al numero massimo di istruzioni.
Il programma è scritto in C e si articola in due sezioni: nella prima si connette alla mappa mentre nella seconda si esaminano e si cancellano eventuali valori.

La mappa, pinnata in memoria, è stata creata dai programmi in XDP oppure in TC-egress: questa soluzione permette al programma in userspace di accedervi, senza doverle allocare, semplificando notevolmente il codice.
La prima porzione, addetta al caricamento della mappa, si serve della funzione OPEN_BPF_MAP_FILE() per connettersi. Questa procedura locale infatti sfrutta l'helper BPF_OBJ_GET_INFO_BY_FD() che tramite una struttura di tipo STRUCT BPF_MAP_INFO ed il file descriptor (ricavato dalla funzione BPF_GET_OBJ()) permette di recuperare tutte le informazioni disponibili della mappa. Avendo a disposizione i dati acquisiti, si procede quindi ad un controllo sulla loro correttezza mettendoli a confronto con una variabile denominata map_expect la quale contiene la struttura della mappa ritenuta esatta.
La seconda parte del programma invece è rappresentabile come un secondo thread che include un ciclo infinito, il quale viene attivato ogni venti secondi(costante TIME_TO_SLEEP, disponibile nel file "constants.h").

```
while (1) {
  k = nk;
  nsec = get_nsecs();
  while(bpf_map_get_next_key_and_delete(map_fd, &k, &nk, &delete,
   counter_map_fd) == 0) {
    k = nk;
    res = bpf_map_lookup_elem(map_fd, &k, &v);
      if(res < 0) {
          printf("No value??\n");
          continue;
      }
      if((nsec - v.age) >= TIME_TO_DELETE) {
    strcpy(ips, inet_ntoa( *(struct in_addr*)&k.saddr));
    strcpy(ipd, inet_ntoa( *(struct in_addr*)&k.daddr));
```

```
14            printf("Deleting key\n\t\tproto:%d\n\t\tip src:%s\n\t\tip
    dst:%s\n\t\tport src:%d\n\t\tport dst:%d\t\n", k.proto, ips,
   ipd, k.sport, k.dport);
15          delete++;
16       }
17     }
18 }
```

Listing 4.7. Funzione addetta all'eliminazione dei dati

Come si può notare dalla porzione di codice 4.3.3, ad ogni attivazione, si recupera il tempo in nano secondi dall'avvio del dispositivo sfruttando la funzione get_nsecs rappresentata in 4.3.3.

```
1 static unsigned long get_nsecs(void)
2 {
3     struct timespec ts;
4     clock_gettime(CLOCK_MONOTONIC, &ts);
5     return ts.tv_sec * 1000000000UL + ts.tv_nsec;
6 }
```

Listing 4.8. Funzione get_nsecs

Potendo disporre del tempo di avvio del sistema, è possibile sfruttare il campo age della struttura valore all'interno della mappa xdp_map per valutare da quanto tempo la entry è stata creata. Utilizzando l'helper BPF_MAP_DELETE_ELEM() si può rimuovere una chiave dalla mappa nel caso si sia rivelata troppo vecchia. Si è dovuti scendere ad un compromesso: se si fossero cancellate troppo spesso, l'accelerazione del traffico avrebbe perso di significato e si sarebbe sovraccaricato il processore facendo passare un traffico elevato nel kernel; in caso contrario i dati sarebbero rimasti i medesimi per troppo tempo. Si è concluso che un minuto trascorso dalla creazione della entry prima della sua eliminazione sia un valore accettabile (TIME_TO_DELETE in "constants.h"). Servirsi di questo macchinismo permette un approccio reattivo alle modifiche all'interno del sistema: il primo pacchetto che

transiterà dopo l'eliminazione attraverso il programma posizionato in XDP, verrà trasmesso al kernel Linux e non verrà accelerato, aggiornando così i dati memorizzati.

Come rappresentato nel codice, si sfrutta la chiamata locale alla funzione BPF_MAP_GET_NEXT_KEY_AND_DELETE(): questa procedura utilizza l'helper BPF_MAP_GET_NEXT_KEY() per navigare attraverso tutte le entry nella mappa, eliminando, se necessario, la chiave precedente. Si è adottato questo criterio perché, una volta eliminata la chiave, sarebbe mancato il riferimento alla successiva causando un mancato accesso alla memoria. Il flag "delete" permette di segnalare che una entry deve essere cancellata. Assegnandogli un valore non nullo e tenendo traccia della struttura key precedentemente usata, si riesce ad evitare questo problema.

```
1 if (*delete) {
2
3     bpf_map_delete_elem(fd, key);
4     *delete= 0;
5   }
6
7 return res;
```

Listing 4.9. Porzione di codice che elimina i dati

Riassumendo: il ciclo di controllo in un thread separato dal flusso principale parte ogni venti secondi, eliminando le entry più vecchie di un minuto, garantendo così un aggiornamento globale e periodico del sistema.

4.3.4 Implementazione calcolo statistiche

Il sistema di routing così costruito accelera i traffici UDP e TCP transitanti nel dispositivo. Uno sviluppo che può risultare utile all'amministratore del sistema è essere a conoscenza della quantità di pacchetti che attraversano il router: nasce

così l'esigenza di dover accedere a delle statistiche sui pacchetti gestiti dal sistema eBPF definito precedentemente. Questa raccolta di dati è eseguita valutando due variabili: il numero di pacchetti che transitano e la dimensione totale di dati trasmessi in bytes.

Questa feature è stata implementata sperimentando tre approcci differenti allo scopo di mantenere le prestazioni e il corretto funzionamento del router sviluppato precedentemente.

Sono sorte due problematiche durante lo svolgimento di questa sezione: la prima è la gestione della concorrenza tra processi differenti che accedono ad una zona di memoria condivisa. La seconda, invece, rappresenta un limite tecnico di eBPF che verrà presentato in seguito 4.3.4.

La gestione della concorrenza, fino a questo momento, era stata stato possibile evitarla a causa del fatto che si avevano solamente due processi: un produttore (programma in TC-egress che scriveva nella mappa) ed un consumatore (programma in XDP che leggeva). La sezione critica, individuata dall'inserimento di una nuova entry, era protetta dall'helper BPF_MAP_UPDATE_ELEM() il quale compie un'operazione atomica (non interrompibile da altri processi) per l'aggiunta. Questa soluzione risulta non più sufficiente nel momento in cui anche il programma in XDP deve aggiornare i campi della mappa. Ciò non è sostenibile perché l'utilizzo del suddetto helper ad ogni passaggio di un pacchetto sarebbe troppo dispendioso in termini di consumo della CPU. Per eludere questo problema, si è scelto di proteggere ed aggiornare solamente i dati necessari alle statistiche: ovvero il contatore dei pacchetti e il calcolo dei bytes totale. Sfruttando il valore di ritorno della funzione bpf_map_lookup_elem(), eseguita nella fase di inoltro del programma in XDP 4.3.2, si può accedere direttamente ai dati ricevuti, in quanto si otterranno il puntatore al valore desiderato. La soluzione adottata è quella di utilizzare la funzione atomica ___SYNCH_FETCH_AND_ADD(): questa procedura permette di aggiungere un valore specificato ad una variabile, entrambi passati come parametri

[17].

Il secondo problema sorto durante l'inserimento del calcolo delle statistiche è insito nella versione del kernel di Linux usata dal dispositivo, ovvero la 5.4.3, ed è dovuto all'utilizzo delle memorie di tipo PER_CPU. All'interno di suddetta distribuzione e nelle precedenti veniva utilizzata la funzione PCPU_COPY_VALUE()[18]: questa procedura veniva chiamata una volta effettuata l'eliminazione di una entry. Concettualmente, la pcpu_copy_value si occupa solamente di rendere "irraggiungibili" quelle zone di memoria di cui si è decretata la cancellazione ma senza azzerarne completamente i valori e rendendoli comunque riusabili dal sistema eBPF. Quest'ultimo le faceva risultare disponibili per una nuova sessione in ingresso. Ciò causava problemi nella scrittura: eseguito il salvataggio da parte di un singolo processore nella propria mappa PER_CPU, i dati nelle entry create per le altre CPU risultavano "sporchi" ed errati. Questo difetto è stato risolto con l'avvento della distribuzione 5.10 e successive, le quali presentavano la funzione PCPU_INIT_VALUE()[19] che si occupava anche di azzerare i valori interni dopo l'eliminazione di questi ultimi. Questa procedura è stata introdotta manualmente nel kernel 5.4.3, adattandolo per garantire la gestione corretta delle mappe. Controllando, nel programma in XDP, il numero di pacchetti salvato per ogni sessione distinta all'interno del singolo processore (deve risultare diverso da 0, in quanto la pcpu_init_value() inizializza tutti i dati a valori nulli), è possibile individuare le entry non ancora utilizzate, facendo passare il traffico attraverso il kernel per riscriverle al TC-egress.

Unica mappa globale per routing e statistiche

La prima soluzione implementata è stata l'integrazione delle statistiche all'interno della mappa già esistente delle variabili addette alle statistiche.

La definizione della struttura _value rappresentata in 4.3 viene modificata aggiungendo i due contatori.

```
typedef struct fields{
    __u8 tos;
    __be16 h_vlan_TCI;
    __u32 ifindex;
    unsigned char    h_dest[ETH_ALEN];
    unsigned char    h_source[ETH_ALEN];
    __u64 age;
    __u64 counter;
    __u64 bytes;
} _value;
```

Listing 4.10. Struct _value modificata con l'aggiunta dei contatori

Il sistema così definito subisce delle alterazioni nel comportamento, in quanto il programma in TC-egress deve inizializzare i contatori quando crea la mappa e quello in XDP deve utilizzare la funzione ___synch_fetch_and_add() per incrementarli. Relativamente al conteggio dei pacchetti è sufficiente incrementarlo di 1 mentre per quello adibito alla quantità totale dei dati transitati è necessario aggiungere la dimensione del pacchetto singolo.
Quanto è stato effettuato è identificabile nella porzione di codice 4.3.4, eseguita appena precedentemente all'inoltro del pacchetto 4.3.2.

```
__sync_fetch_and_add(&ret->counter, 1);
__sync_fetch_and_add(&ret->bytes, (data_end-data));
```

Listing 4.11. Modifica dei contatori

Questo approccio presenta dei vantaggi: infatti, permette di utilizzare una sola mappa facendo così un solo lookup, riducendo il tempo necessario per accedere in memoria.
Presenta però degli svantaggi notevoli: sfruttando una sola zona di memoria condivisa risulta che i vari processori dovranno aspettare che un'operazione di scrittura eseguita da un core (come può essere l'aggiornamento dei campi delle statistiche) sia

conclusa prima di poter scrivere a loro volta. Ciò aumenta notevolmente il tempo di esecuzione del singolo pacchetto dovuto al tempo di attesa dei singoli processori per poter scrivere in una mappa condivisa. Un'ulteriore svantaggio di questa soluzione è simbolico ed è rilevabile nella perdita di importanza delle statistiche: dovendo cancellare periodicamente le entry della mappa per rinnovare le tabelle di routing, si eliminerebbero anche i dati relativi ai contatori rendendoli pressoché privi di valore.

Unica mappa PER-CPU per routing e statistiche

La seconda soluzione adottata è quella di utilizzare una singola mappa di tipo BPF_MAP_TYPE_PERCPU_HASH, in modo da eliminare l'attesa riguardante la scrittura concorrente. La struttura _value risulta costituita in modo equivalente a 4.3.4 in quanto cambia solamente la definizione e l'allocazione della mappa. Il comportamento adottato dai vari programmi in kernel space, infatti, è il medesimo di quanto visto nella sezione 4.3.4.
L'unica sostanziale alterazione del codice è localizzata nel programma entry-cleaner.c. Infatti, la funzione bpf_map_lookup_elem() eseguita su una mappa definita PER_CPU, ritorna un puntatore ad un vettore contenente, per ogni cella, il valore stabilito da ciascuna CPU per una determinata chiave. Ad ogni ricerca dovrà essere esaminato un vettore di lunghezza pari al numero delle CPU: di questi dati memorizzati sarà sufficiente una sola entry troppo datata (confrontando il campo age della struttura _value) per segnalare come eliminabile la chiave. La funzione delete_entry() visualizzata in 4.3.3 verrà arricchita esaminando i quattro valori (nel caso della scheda LS-1046A-FRWY).

```
1    for(i = 0; i < NUM_CPU && !flag; i++)
2      if((nsec - values[i].age) >= TIME_TO_DELETE) {
3    strcpy(ips, inet_ntoa( *(struct in_addr*)&k.saddr));
4    strcpy(ipd, inet_ntoa( *(struct in_addr*)&k.daddr));
```

```
5           printf("Deleting key\n\t\tproto:%d\n\t\tip src:%s\n\t\tip
    dst:%s\n\t\tport src:%d\n\t\tport dst:%d\t\n", k.proto, ips,
   ipd, k.sport, k.dport);
6       delete++;
7       flag++;
8       }
```

Listing 4.12. Funzione delete_entry modificata per esaminare i dati in ogni singola CPU

Sarà necessario sommare i valori ottenuti per avere la statistica completa. Questo calcolo viene compiuto nel momento precedente all'eliminazione dei dati. Dal confronto fra la porzione di codice 4.3.4 e 4.3.3 emerge che viene compiuta solo una semplice operazione matematica per avere la somma totale dei contatori.

```
1 if (*delete) {
2     for(i = 0; i < NUM_CPU; i++){
3       sum_p += v[i].counter;
4       sum_b += v[i].bytes;
5     }
6     bpf_map_delete_elem(fd, key);
7     printf("\tTotal packet per session: %d\n\tTotal bytes per
   session: %d\n", sum_p, sum_b);
8     *delete= 0;
9 }
```

Listing 4.13. Somma dei valori di ogni CPU

Questa soluzione, come si noterà nella sezione dedicata ai risultati, semplifica il costo delle attese riguardanti l'accesso alle memorie condivise, a scapito dell'aumento del traffico passante attraverso il kernel. Questo avviene perché è impossibile sincronizzare tra di loro le varie CPU: avendo ogni processore una entry dedicata, non potrà accedere alle altre, quadruplicando (è una crescita lineare rispetto alla soluzione precedente) il numero di pacchetti gestiti dal sistema operativo invece di essere accelerati.

Riguardo l'attendibilità e il valore dei dati raccolti, risulta lo stesso problema della sezione 4.3.4 in quanto aggiornando la tabella di routing, si azzerano anche le statistiche.

Unica mappa per routing, mappa PER-CPU per statistiche

Il terzo metodo per introdurre le statistiche nel sistema di accelerazione consiste in un approccio misto alle due soluzioni tentate in precedenza. Si utilizzano due mappe distinte: una di tipo BPF_MAP_TYPE_HASH per la gestione del routing ed una BPF_MAP_TYPE_PERCPU_HASH per le statistiche. Questa tecnica è basata sul fatto che la prima subisce molti meno aggiornamenti rispetto all'altra, per cui assume un'importanza strategica notevole separarle.
Sfruttando sempre la coppia chiave/valore anche per la seconda mappa, chiamata COUNTER_MAP, diviene necessario ridefinire la struttura che contiene i contatori relativi al singolo flusso.

```
typedef struct counter_fields{
    __u64 counter;
    __u64 bytes;
} _counter_value;
```

Listing 4.14. Struttura dati che include i contatori

In questo modo, tutti i programmi creati subiscono dei cambiamenti: essendo disaccoppiati statistiche e routing, in tc2.c (TC-egress) è necessario verificare che la entry non esista prima di effettuare una bpf_map_update_elem(). In caso sia già presente è sufficiente utilizzare le __synch_fetch_and_add per risparmiare sul consumo di CPU. Per effettuare questo controllo, si sfrutta l'helper bpf_map_lookup_elem(). Il programma posizionato in XDP, a sua volta, effettua i medesimi controlli eseguendo le operazioni dualmente, come emerge dalla porzione di codice seguente:

```
ret_cv = bpf_map_lookup_elem(&counter_map, &_key);
if(ret_cv != NULL){
    __sync_fetch_and_add(&ret_cv->counter, 1);
    __sync_fetch_and_add(&ret_cv->bytes, (data_end-data));
}else{
    cv.counter = 1;
    cv.bytes = (data_end-data);
    bpf_map_update_elem(&counter_map, &_key, &cv, BPF_ANY);
}
```

Listing 4.15. Aggiornamento statistiche XDP

Il programma in userspace, invece, deve essere collegato a due mappe differenti ed eseguire due volte i controlli sulla loro corretta formattazione (come già affrontato in 4.3.3). Successivamente, nella sezione di codice in cui viene eliminata la entry, si sommano le statistiche memorizzate in ogni processore similmente a come veniva fatto in 4.3.4, con l'aggiunta di una seconda lookup sulla mappa dedita alla raccolta dei dati.

```
if (*delete) {

  bpf_map_delete_elem(fd, key);
  bpf_map_lookup_elem(counter_fd, key, cv);
  for(i = 0; i < NUM_CPU; i++){
    sum_p += cv[i].counter;
    sum_b += cv[i].bytes;
  }
  printf("\tTotal packet per session: %d\n\tTotal bytes per session
   : %d\n", sum_p, sum_b);
  *delete= 0;
}
```

Listing 4.16. Somma dei valori di ogni CPU con lookup precedente

Il vantaggio di questa soluzione è il disaccoppiamento dei dati dalla mappa dedita al routing. Questo è un vantaggio notevole perché, per quanto la seconda mappa

si possa aggiornare, il calcolo delle statistiche verrà effettuato indipendentemente dalla presenza o meno della entry nella xdp_map. La tecnica utilizzata ha un costo elevato dal punto di vista computazionale perché occorreranno due lookup in due mappe differenti ad ogni pacchetto gestito dal programma in XDP.
Un ulteriore aggiornamento che può essere introdotto è una modifica al tipo di mappa adibita ai contatori definendola come BPF_MAP_TYPE_LRU_PERCPU_HASH. Specificandola in questo modo, invece della più classica BPF_MAP_TYPE_PERCPU_HASH, si aggiunge la specifica LRU alla tabella di tipo hash: questo significa che, in caso di mappa piena, verrà eliminata la entry usata meno di recente[20].

4.4 Automazione dei servizi

Per rendere tutto il sistema automatico, è stata sfruttata una suite di comandi per la gestione dei servizi di Linux: SYSTEMD. Il principale comando per l'utilizzo di systemd è *systemctl*. Permette inoltre l'uso di diverse tipologie di unità che garantiscono l'inizializzazione e supervisionamento dell'intero sistema: tra quelle disponibili sono stati utilizzati i servizi, ovvero demoni che permettono la creazione, il riavvio ed il ricaricamento di programmi. Altri tipi di unità gestibili con systemd sono i mountpoint o i socket generici [21].
Utilizzando questa suite di comandi sono stati sviluppati due servizi che gestiscono la creazione del sistema di accelerazione che si basa su eBPF. Si è scelto di realizzarne due differenti perché gli aspetti da gestire sono diversi a causa dell'omogeneità di programmi da coordinare. Utilizzando il comando systemd, sono entrambi avviati durante la fase di boot del dispositivo.
Il primo servizio, chiamato "router-xdp.service", permette la compilazione ed il caricamento dei programmi in XDP e in TC-egress. Il servizio esegue un file bash ("attach.sh") che contiene un ciclo a contatore che permette di collegare i programmi a tutte le interfacce: queste sono individuabili attraverso l'uso di un'espressione

regolare eseguita sul risultato del comando da console "ip link".
Il file che contiene il programma in userspace è compilato anche all'interno del file bash eseguito da questo servizio.

```
[Unit]
Description=Traffic Accelerator XDP

[Service]
User=root
WorkingDirectory=/xdp-router
ExecStart=/bin/bash attach.sh

[Install]
WantedBy=multi-user.target
```

Listing 4.17. Servizio router-xdp.service

Il secondo servizio, rinominato "entry-cleaner.service", permette la gestione del programma in userspace. Quest'ultimo è un file eseguibile, al contrario dei programmi in XDP e in TC che risultavano essere dei file oggetto. Inoltre, gestendo un ciclo infinito, questo servizio deve rimanere attivo fino allo spegnimento del dispositivo. router-xdp.service, al contrario, doveva eseguire il file con estensione .sh per poi terminare. Il programma entry-cleaner.c si collega alle mappe senza allocarle, compito dedicato a quello in XDP o in TC-egress: questa operazione potrebbe causare dei rallentamenti. Per risolvere questa problematica, si deve specificare un ordine di precedenza all'interno dei servizi (indicando che "entry-cleaner.service" debba essere eseguito successivamente a "router-xdp.service") e, nel caso questo non basti, tentare più volte l'avvio del programma in userspace.

```
[Unit]
Description=Entry Cleaner Service
After=router-xdp.service

```

```
[Service]
User=root
ExecStart=/xdp-router/entry_cleaner
Restart=on-failure
KillMode=process
Type=simple
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Listing 4.18. Servizio entry-cleaner.service

Capitolo 5

Valutazione sperimentale

Sono stati effettuati diversi tipi di test per valutare differenti caratteristiche del sistema. Si sono misurate le performance in termini di consumo della CPU e quale throughput riuscisse a mantenere il dispositivo. Si è inoltre testato il sistema sia con traffico UDP che con TCP per verificare come l'apparato avrebbe retto al variare delle diverse configurazioni del generatore di traffico.
L'intero sistema è stato testato sulla piattaforma di sviluppo LS1046A-FRWY le cui specifiche sono trattate nella sezione 5.1.1.

5.1 LS1046A Freeway Board

La scheda LS1046A Freeway Board (o anche LS1046A-FRWY), prodotta da NXP, azienda proveniente dai Paesi Bassi, è una piattaforma di elaborazione, valutazione e sviluppo ad alte prestazioni. Ha a disposizione, oltre ad un modulo WiFi, anche un acceleratore per applicazioni di tipo di Machine Learning e Artificial Intelligence.
È stata utilizzata durante questo lavoro di tesi come dispositivo su cui costruire l'infrastruttura di accelerazione. Questo ha portato ad affrontare delle problematiche notevoli, tra le quali spiccano la costruzione dell'ambiente di sviluppo 6.1

e la modifica al kernel necessaria per evitare che la struttura *struct skb** venisse frammentata dal programma XDP 5.3.1. Entrambi i problemi sono stati trattati in seguito.

5.1.1 Caratteristiche tecniche

1. **Processore**: LS1046A - 4 Core ARM @1.6GHz
2. **Elementi di Networking**:
   - 2x M.2 slots – Wi-Fi, 4G/LTE, SSD
   - 4x 1GE Ethernet
   - Supporti per espansioni di tipo Mikro-Click
3. **Periferiche di base**:
   - 4GB DDR4 con ECC (Error Correcting Code) @ 2.1GT/s
   - 4Gb NAND FLASH
   - 64MB QSPI FLASH
   - 2x USB3.0 (host)
   - 1x slot MicroSD
4. **Caratteristiche aggiuntive**:
   - Prevede degli header aggiuntivi per il supporto di Clock, GPIO o Interrupt

5.2 Generatori di traffico

Un generatore di traffico è un dispositivo in grado di produrre traffico in maniera controllata attraverso la rete. Si definisce "controllata", nel senso che rispetta i canoni e le caratteristiche decisi dall'utilizzatore per un determinato tipo di test.

Sono utilizzati principalmente per eseguire test di prestazioni e verificare i corretti comportamenti di una rete. In seguito si esamineranno più nel dettaglio i generatori di traffico utilizzati.

5.2.1 Cisco TREX

TRex è uno strumento di generazione del traffico open source veloce e realistico, in esecuzione su processori Intel standard, basato su DPDK. Supporta le modalità di generazione del traffico stateful e stateless. Una delle sue peculiarità è l'estrema velocità con cui riesce a raggiungere le specifiche definite dal test [22]. TRex fornisce inoltre una GUI per configurare in modo semplice e veloce i test. Nello sviluppo di questo progetto è stato utilizzato solamente tramite una CLI.
Con un semplice programma scritto in Python, TRex permette di gestire e configurare la prova da effettuare. Per l'assegnazione degli indirizzi IP e dei MAC relativi alle interfacce, è necessario compilare un file YAML in modo facile e veloce.
Per l'esecuzione di TRex in modalità server è necessario utilizzare il comando

```
./t-rex-64 -i -c 14
```

Listing 5.1. Comando esecuzione TRex in modalità server

L'opzione *-i* permette di eseguire il server in modalità interattiva, quindi potendo collegarsi tramite una console. L'opzione *-c [N]* permette invece di stabilire un numero di core da sfruttare per generare il traffico desiderato.
Questo generatore di traffico fornisce inoltre le statistiche cercate senza dover utilizzare ulteriori tools. Dall'immagine 5.1 si può notare la confurazione sfruttata per eseguire il test: il generatore di traffico sfruttava due porte a cui erano assegnati degli indirizzi IP statici e sul router venivano configurate, oltre agli indirizzi, anche le rotte necessarie all'inoltro del traffico.

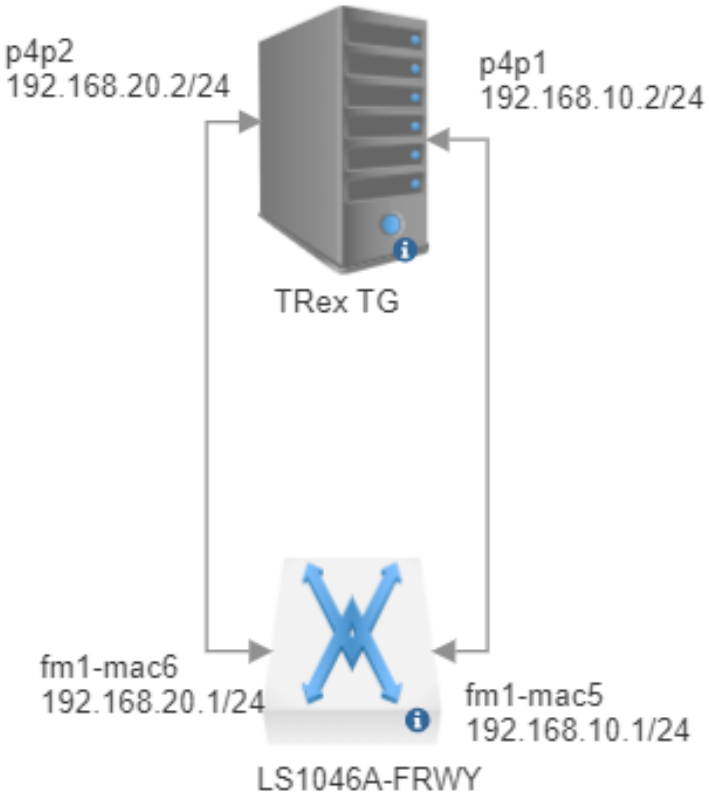

Figura 5.1. Configurazione test TRex UDP

5.2.2 IxLoad-Keysight

Per quanto riguarda il secondo generatore di traffico utilizzato: si tratta di IxLoad, un prodotto di Ixia, azienda specializzata nella costruzione di dispositivi per test acquistata nel 2015 da Keysight.

IxLoad offre test completi delle prestazioni per convalidare la qualità dell'esperienza dell'utente e funziona emulando web, video, voce, storage, VPN, wireless, infrastruttura e protocolli di incapsulamento/sicurezza per creare scenari realistici. Un design del sistema modulare consente a IxLoad di scalare con l'infrastruttura, mentre le metriche di qualità dell'esperienza in tempo reale consentono di approfondire per identificare rapidamente i guasti della rete e isolare i punti di rottura [23].

5.3 Confronti con approcci precedenti

In questa sezione verranno valutate le prestazioni del sistema costruito sfruttando la tecnologia eBPF. Sono stati effettuati due tipi di test per valutare le performance in caso di traffici diversi: nel primo si è sfruttato TRex per una prova sotto sforzo del dispositivo andando a verificare il numero massimo di pacchetti gestibili, nel secondo si è esaminato il comportamento dell'apparato all'aumentare del numero di sessioni differenti utilizzando IxLoad.

Per entrambi si è sfruttato un approccio simile: il generatore di traffico veniva collegato tramite due interfacce a due porte dell'apparato sotto esame in modo da poterne misurare le statistiche.

Il traffico UDP è stato gestito in modo bidirezionale: i pacchetti venivano trasmessi dall'interfaccia p4p1 alla p4p2 e viceversa transitando attraverso la scheda, raddoppiando così il numero di dati passanti. Si può dire che le porte si comportavano in modalità peer.

Invece, il traffico TCP, dovendo sfruttare più sessioni diverse, è stato gestito interamente da IxLoad con un approccio client/server delle porte: su un'interfaccia dell'infrastruttura Ixia veniva caricato un server HTTP, sull'altra un client che faceva richiesta di pagine web, il dispositivo LS1046A-FRWY si comportava invece come un comune router.

I primi di questi test, eseguiti generando un traffico di tipo UDP, sono stati effettuati con l'ausilio del generatore di traffico TRex. Sono state effettuate due tipi di prova: nella prima si verificava il consumo dei processori facendo il confronto con le tecnologie precedenti, nell'altra si esaminava l'impatto dei contatori sul sistema generale.

Durante la prima prova si sono messi a confronto tre soluzioni radicalmente diverse: la prima consiste nella verifica delle prestazioni della macchina base, ovvero senza nessun tipo di accelerazione dei dati. La seconda tecnologia è stata compiuta valutando il funzionamento di una scheda LS1046A-FRWY che sfuttasse FlowOffload,

ovvero un ottimizzatore di netfilter. Per ultimo, si sono misurate le performance dell'apparato utilizzante l'acceleratore basato su eBPF costruito durante questo lavoro di tesi.

5.3.1 Risultati

I test sono stati eseguiti valutando due principali variabili: il throughput della rete e il consumo medio delle CPU del dispositivo. Il primo è stato fornito da TRex (più precisamente dalla console di TRex che presentava inoltre il numero di pacchetti e bytes trasmessi), mentre per la seconda misura si è utilizzato *mpstat*, un programma Linux che è possibile lanciare da linea di comando e che stampa a video le prestazioni medie in un intervallo di tempo di un determinato processore. Mpstat è stato lanciato utilizzando il comando

```
1 mpstat -P ALL 2
```

Listing 5.2. Comando esecuzione mpstat

L'opzione *-P* permette di selezionare i processori da monitorare. Invece, il numero successivo indica l'intervallo di tempo su cui eseguire la media delle valutazioni [25]. Si sono sfruttate due delle quattro porte a 1Gbps disponibili sulla scheda andando a variare la dimensione dei pacchetti lasciando stabile la velocità di trasmissione. Il traffico UDP permette di gestire pacchetti fino a 64B, questo per mettere sotto stress il dispositivo.

Test throughput

Dalla figura (5.2) si può notare come la macchina senza alcun tipo di accelerazione non riesca a raggiungere un throughput sufficiente per trasmettere tutti i pacchetti perdendone molti. Calcolando un rapporto tra pacchetti ricevuti e pacchetti trasmessi, si può notare che, senza ottimizzazioni, il dispositivo perde circa il 48% dei dati. Questo è dovuto al fatto che tutti i dati devono transitare attraverso il kernel

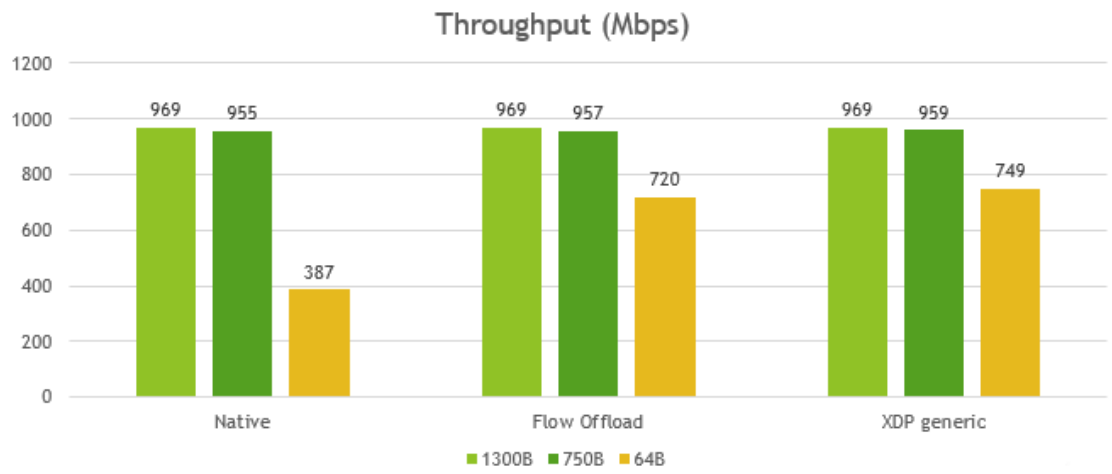

Figura 5.2. Throughput traffico passante

del sistema operativo, rallentando notevolmente il loro passaggio tramite l'apparato.

Relativamente a FlowOffload e il sistema costruito con l'utilizzo di eBPF, si può verificare come le due soluzioni si comportino in maniera simile al diminuire della dimensione dei pacchetti: questo perché, in entrambi i casi, non risultano perdite.

Test CPU

Per quanto riguarda il consumo di CPU si possono notare degli schema ricorrente. Infatti, il numero di pacchetti che i processori riescono a gestire cambia notevolmente al variare delle soluzioni.

Si nota che, già con pacchetti di grandi dimensioni, la macchina senza ottimizzazioni presenta già un idle del processore notevolmente più basso rispetto agli altri due approcci. Focalizzando l'attenzione su quella soluzione, si evidenzia un calo notevole di performance, quando la dimensione dei pacchetti varia da 400B a 100B, arrivando ad azzerarsi totalmente al raggiungimento di quest'ultima soglia.

FlowOffload presenta caratteristiche migliori rispetto alla macchina "bare metal"

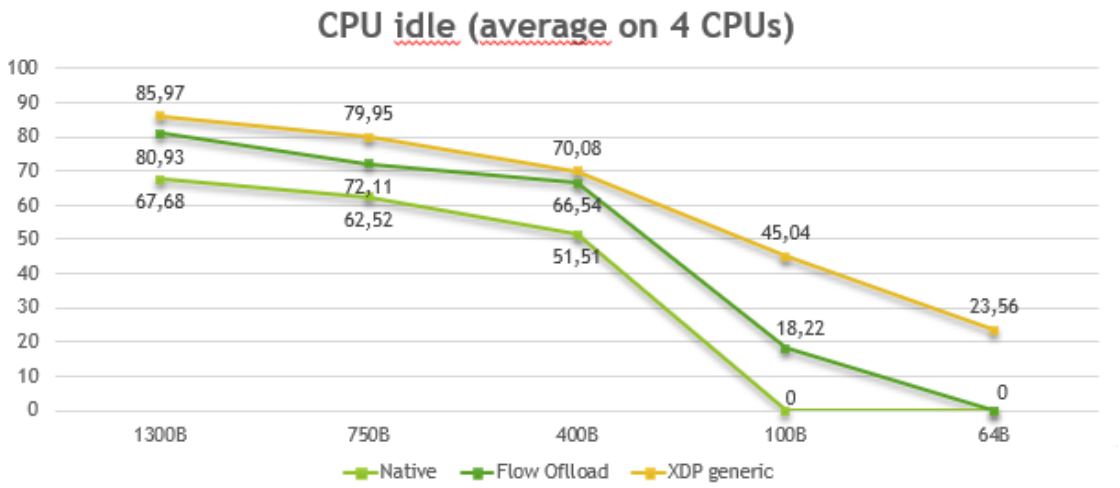

Figura 5.3. Consumo medio delle CPU

in quanto il traffico transita attraverso gli hookpoint di Netfilter che eseguono un lavoro ottimizzato.

Come si è visto precedentemente, il throughput risulta simile a quello della soluzione eBPF, ma, da quello che risulta analizzando il comportamento dei singoli processori, si scopre che presenta un CPU idle tendente allo 0%. Da ciò si evince che non presenta ulteriore "spazio di manovra" per l'aggiunta di ulteriori servizi avendo tutti i core completamente occupati. Un'ulteriore curiosità che si può notare è che il consumo cala in maniera simile a quello della macchina senza accelerazioni. Questo succede perché, essendo solo un'ottimizzazione, il percorso dei pacchetti è lo stesso della macchina base.

Questo non succede invece nel caso del sistema costruito sfruttando la tecnologia eBPF. Come si nota, la percentuale di consumo dei processori aumenta (l'idle delle CPU diminuisce) gradualmente, non annullandosi nemmeno alla minima dimensione dei pacchetti trasmessi (64B, per garantire la presenza di un header Ethernet). Questo significa che, a parità di traffico trasmesso, la soluzione presentata durante questo lavoro di tesi risulta la migliore tecnica tra quelle esaminate.

È inoltre stato necessario ampliare il parametro skb_headroom del sistema, in

quanto risultava troppo piccolo per poter utilizzare XDP: quest'ultimo richiede almeno una dimensione di 256B mentre il dispositivo forniva solamente 16B. Modificando i bootargs, ovvero le variabili di sistema che vengono caricate all'avvio, si è aumentato il valore dell'skb_headroom a 272B (ovvero i 16B già predisposti sommati alla dimensione minima richiesta). Senza questo cambiamento, l'apparato non riusciva a caricare la struttura skb in memoria in un solo passaggio, facendo così due accessi alla memoria rallentando notevolmente il sistema.

Test CPU con statistiche

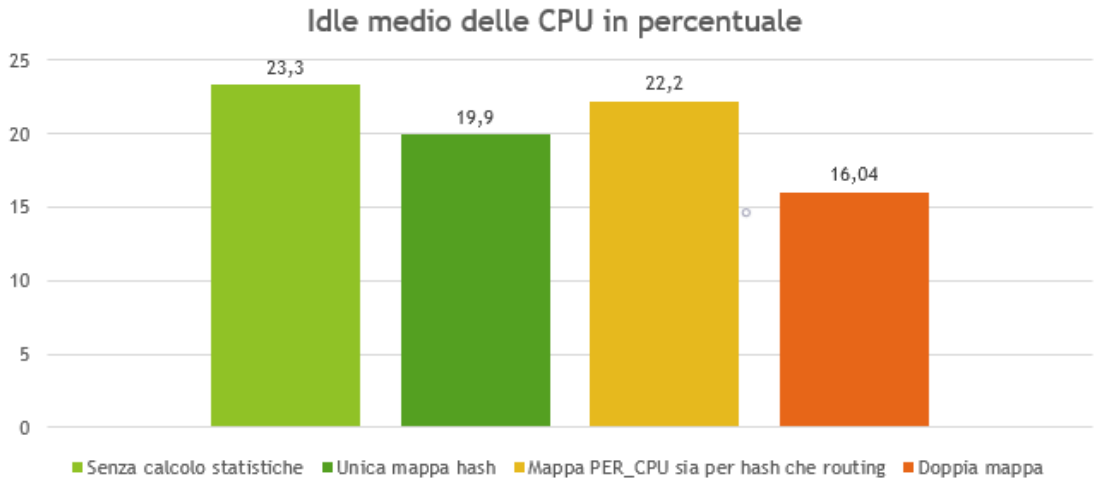

Figura 5.4. Consumo medio delle CPU con statistiche

si evince come incida in modo importante l'implementazione delle statistiche, specialmente sull'implementazione che utilizza la doppia mappa (4.3.4). Questo è dovuto alla doppia chiamata all'helper BPF_MAP_LOOKUP_ELEM(): la prima volta viene utilizzato per cercare la entry nella tabella di routing e successivamente anche in quella dei contatori per incrementarli. Questa doppia ricerca pesa in modo notevole, rispetto alle altre soluzioni, sul carico di lavoro di ogni processore.
Il rallentamento dell'infrastruttura, costruita con una sola mappa di tipo hash

(4.3.4), è invece dovuto all'attesa introdotta dalla funzione atomica ___SYNCH__FETCH__AND__ADD()
Come già accennato, ogni CPU dovrà attendere il proprio turno, in attesa che i contatori siano stati incrementati e l'operazione conclusa dagli altri. Questo implica una sospensione dell'esecuzione delle istruzioni successive (tra cui figura anche l'inoltro del pacchetto stesso).
La terza soluzione da trattare è quella che utilizza un'unica mappa PER_CPU sia per il routing che per le statistiche (4.3.4). Questo approccio elimina sia l'attesa per la scrittura concorrente, sia l'esigenza di dover cercare in due mappe distinte. Come già trattato, utilizzando questo metodo, verrà meno l'importanza dei contatori in quanto saranno continuamente azzerati.

5.4 Scalabilità con sessioni multiple

Il secondo tipo di test è stato effettuato per valutare l'effetto di un numero elevato di entry nella mappa adibita al routing. Questa variabile, raggiunto un numero elevato di flussi, potrebbe rallentare in maniera significativa le ricerche sia nella tabella di routing sia in quella delle statistiche. Per svolgere questo test è stato usato il generatore di traffico IxLoad che, attraverso l'utilizzo di traffico TCP, ha aperto un numero elevato di connessioni HTTP facendo richieste di pagine web.
L'obbiettivo del test figurava nel raggiungere il throughput massimo possibile del dispositivo, ovvero 1Gbps.
IxLoad però non permette di stabilire la dimensione dei pacchetti: risulta essere una variabile che esso stabilisce a priori in base al tipo di obiettivo (nel nostro caso il throughput). I risultati ottenuti non sono paragonabili a quelli delle sezioni precedenti riscontrati con traffico UDP.

5.4.1 Risultati

I test sono stati effettuati paragonando la macchina senza l'utilizzo di acceleratori e le tre configurazioni utilizzate per le statistiche. Come si può notare dalla figura (5.5), il numero di sessioni gestite in tutti i test è stato fisso a circa 21000 per ogni test effettuato.

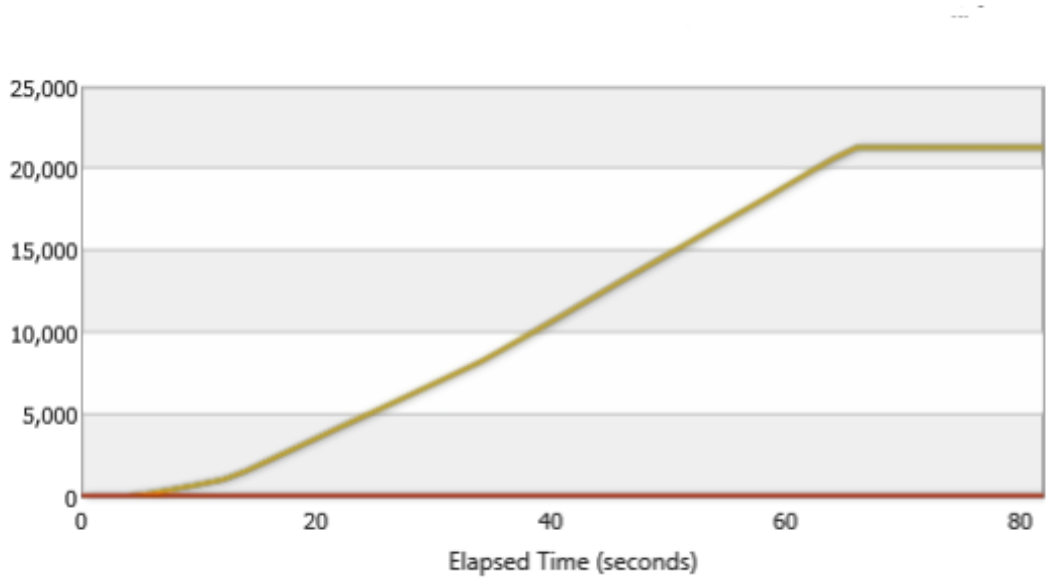

Figura 5.5. Numero di sessioni create totali

Il throughput è stato impostato come obiettivo da mantenere fisso a 1Gbps ed è stato misurato il consumo di CPU medio che il dispositivo impiegava in ognuna delle configurazioni del sistema.
Come si evince dalla figura (5.6), l'apparato riesce a gestire in maniera eccellente il traffico senza alcun tipo di accelerazione, arrivando ad ottenere un idle medio dei processori di circa 63%, questo a causa del fatto che il numero di pacchetti non risultava elevato nonostante il throughput a 1Gbps.
Per quanto riguarda le altre configurazioni: si può notare che esiste uno schema che viene ripreso dalla figura (5.4). Infatti, risulta che la soluzione migliore è una

mappa di tipo PER_CPU unica sia per il routing che per il calcolo delle statistiche: ciò a causa dell'assenza di attese concorrenti a scapito però dell'efficiacia dei dati salvati. Il tutto con un idle medio di 90%.
Relativamente all'approccio che utilizza un'unica mappa di tipo hash (4.3.4), si evidenzia una diminuzione nelle prestazioni esattamente come era stato nel caso di traffico UDP. Per lo stesso motivo, ovvero la presenza di attese concorrenti, questo tipo di approccio è considerato il peggiore tra quelli affrontati.
L'ultimo test effettuato, quello con l'utilizzo di una doppia mappa (4.3.4), ha rispecchiato i valori attesi: infatti ha risolto il problema della concorrenza e della consistenza dei dati, ma ottenendo delle performance meno efficaci rispetto alle soluzioni precedenti. Nonostante questo aspetto, risulta migliore della soluzione che non adotta l'accelerazione.

Con un numero di sessioni stabile a circa 21000 ci si aspettava un decadimento

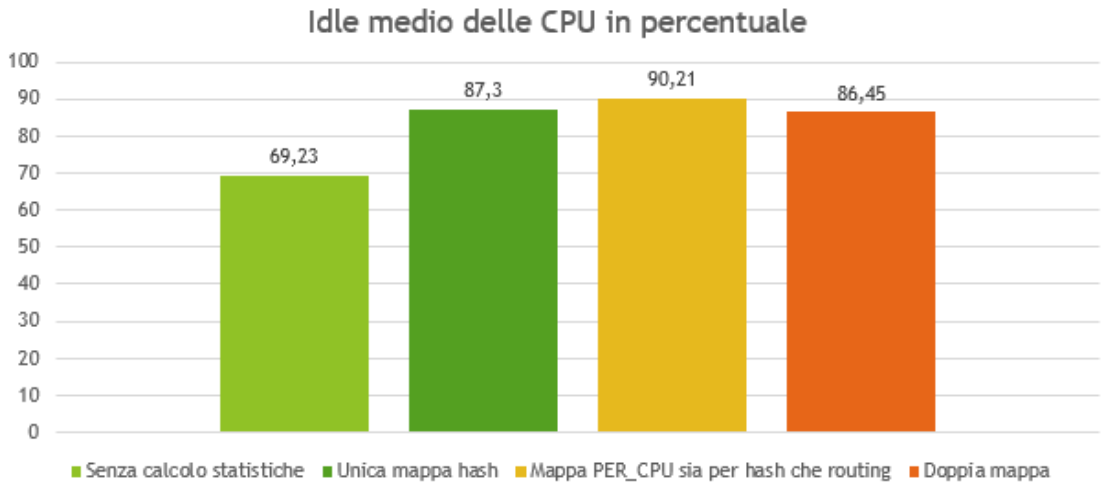

Figura 5.6. Consumo medio delle CPU con statistiche su traffico multi-sessione

delle prestazioni dovuto alla mole di lavoro del programma in userspace. È logico pensare che questo programma abbia un costo significativo per un solo processore, ovvero quello che lo esegue. Gli altri core (almeno nel caso del servizio *fmc.service*

disabilitato) saranno impegnati ad elaborare gli altri flussi di dati (per rimandi, consultare la sezione 3.2.1): a questo proposito, è stato effettuato un test per valutare se la divisione dei pacchetti per sessione su diverse CPU possa influire sulle performance, ma non si sono riscontrate differenze a livello di consumi.
Dalla figura (5.7) si può riscontrare come l'impatto del programma entry-cleaner.c non sia così influente sulle prestazioni del sistema. Infatti si può notare come ci sia una differenza minima (si attesta una differenza media del 2%) tra il sistema con il programma userspace attivo e quello disattivo, nonostante un numero di entry molto elevato. Questo calcolo, con il test eseguito con il generatore di traffico Cisco TRex, non risultava veritiero in quanto permetteva di creare solamente due flussi nella tabella *xdp_map*, ovvero quella che include i dati di routing.

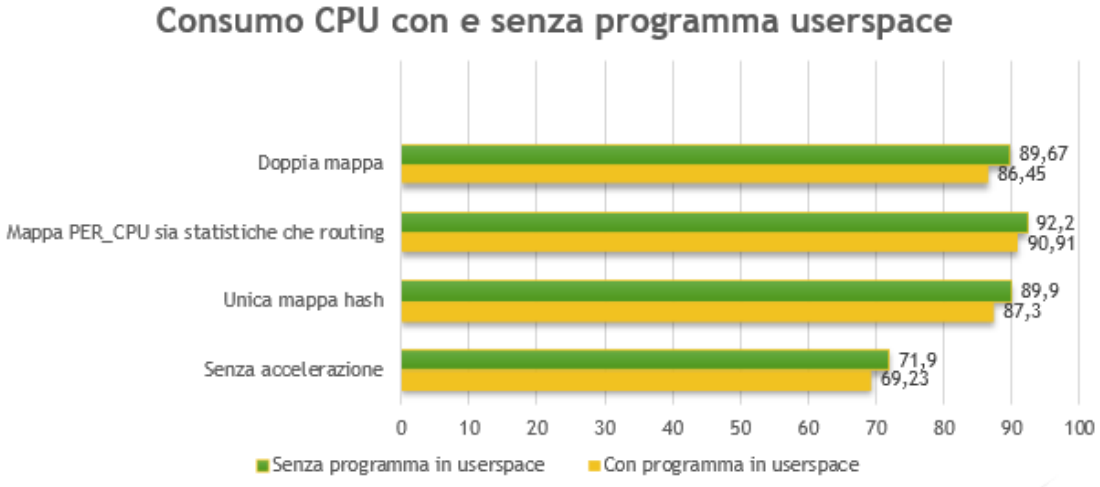

Figura 5.7. Consumo medio delle CPU con e senza programma entry-cleaner.c

Capitolo 6

Conclusioni

L'obiettivo da perseguire, da parte di Tiesse, consisteva nella realizzazione di un sistema di accelerazione software in grado di garantire alte prestazioni a bassi consumi. I vari tentativi precedenti, come quello che ha sfruttato l'ottimizzatore di Netfilter Flow Offload, avevano portato a risultati non pienamente soddisfacenti.
Si sono cercate allora soluzioni alternative, tra le quali è spiccato eBPF. I risultati di un acceleratore hardware, ovvero un chip dedicato, non erano raggiungibili da nessuna soluzione software.
Lo sviluppo di questo progetto di tesi migliorava alla realizzazione di un sistema di accelerazione basato interamente sull'uso della tecnologia eBPF, più nello specifico mirava a sfruttare le caratteristiche dell'hookpoint XDP per alleggerire il carico di lavoro dei vari processori che avrebbero dovuto, altrimenti, elaborare ogni pacchetto all'interno del kernel di Linux.
I risultati ottenuti utilizzando questa tecnologia, facendo cooperare tra loro più programmi differenti, hanno portato a risultati ottimi, anche in vista di sviluppi futuri che potrebbero aumentarne ancora di più le performance. Il confronto con le soluzioni adottate in passato, ha dimostrato come questo approccio sia risultato il migliore (come si può verificare dalla figura 5.3).
Si è dimostrato come la gestione dei singoli pacchetti il più possibile vicino all'interfaccia di ingresso, sia la soluzione migliore per l'accelerazione. I risultati (sezione

5) mostrano come il sistema costruito durante questa tesi sia il più performante dal punto di vista delle prestazioni, ma forse non il più immediato da quello della facilità di manutenzione: essendo un'infrastruttura che utilizza moduli cooperanti tra loro, un singolo cambiamento implica la modifica anche degli altri due programmi per mantenere funzionante la struttura.

6.1 Sviluppi futuri

Il sistema è funzionante con questa configurazione ma è possibile aumentare ulteriormente le performance o, più in generale, fornire servizi aggiuntivi.
Il primo di essi sarebbe l'introduzione nella scheda del supporto per xdp_driver. Questa tecnologia, messa a confronto con quella xdp_generic utilizzata durante lo sviluppo del presente lavoro di tesi, consentirebbe al sistema di raggiungere prestazioni migliori. L'introduzione di questa funzionalità comporta tuttavia la necessità di apportare una modifica al driver della scheda di rete al fine di poterla utilizzare. Caricare i programmi in modalità xdp_driver aumenta le prestazioni dell'hookpoint eBPF di circa il 20-30%. [5]
Un altro possibile miglioramento da sviluppare riguarda lo sfruttamento di una distribuzione di Linux basata su Buildroot. A causa delle difficoltà incontrare nell'installazione e nella costruzione del kernel, l'opzione sopracitata non è stata adottata durante il presente lavoro di tesi preferendole la distribuzione Ubuntu Main che si serviva di *apt* e *apt-get* per l'installazione delle componenti. I dispositivi progettati da Tiesse sfruttano il programma flex-builder per la costruzione del file system e di tutte le componenti per il funzionamento: ciò non è stato possibile a causa di una serie di errori che emergevano nel momento dell'installazione di clang e LLVM, utili per la compilazione e traduzione in bytecode dei programmi eBPF. L'impiego di questa distribuzione potrebbe aumentare ulteriormente le prestazioni dell'infrastruttura.

L'istituzione di ulteriori servizi sarebbe responsabile della diminuzione delle performance dell'apparato ma contribuirebbe a rendere disponibili nuove capacità (vedere riferimenti alla figura 5.3). Tale funzione sarebbe espletata da quei processori che rimanevano liberi da operazioni durante la gestione dei pacchetti, possibilità negata da altri approcci come FlowOffload.
Uno tra tutti, è da considerarsi l'introduzione di un sistema di sicurezza che protegga il dispositivo da traffico malevolo. Un semplice programma di filtering, eseguito nel programma XDP potrebbe risultare essere la soluzione per l'implementazione di un firewall. Infatti, basterebbe aggiungere una nuova mappa o un campo nella tabella di routing per indicare che tale pacchetto potrebbe essere una minaccia per eseguire un'operazione di XDP_DROP e scartarlo.

6.2 Considerazioni finali

Il presente lavoro di tesi, incentrato sul risparmio delle prestazioni in termini di CPU, ha portato ad una soluzione basata sulla tecnologia eBPF che ha dimostrato di essere funzionale allo scopo richiesto. Sono state raggiunte performance inaccessibili attraverso approcci alternativi e, come già visto, sono presenti ulteriori spazi di manovra. Questo tipo di metodo ha garantito inoltre una flessibilità notevole al sistema, ottenuta tramite l'impiego di eBPF.
L'oggetto di lavoro della presente tesi è da preferire per la sua migliore efficienza nonostante la disponibilità di altre soluzioni che garantiscono una più semplice manutenzione del codice (ad esempio, quella che utilizza l'helper bpf_fib_lookup() per accedere alle tabelle del kernel).
In ogni caso, i risultati ottenuti sono da considerarsi incoraggianti in vista di uno sviluppo futuro di dispositivi di routing senza un acceleratore fisico ma solamente software da parte di Tiesse.

Bibliografia

[1] Autori Linux Kernel, Linux Kernel Documentation, *eBPF verifier*, https://docs.kernel.org/bpf/verifier.html

[2] Autori Linux Kernel, Linux Kernel Documentation, *eBPF maps*, https://docs.kernel.org/bpf/maps.html

[3] Linux manual page, *eBPF helpers*, https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[4] Høiland-Jørgensen, Toke, et al. *Proceedings of the 14th international conference on emerging networking experiments and technologies. 2018.*, "The express data path: Fast programmable packet processing in the operating system kernel."

[5] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, *xdp_driver*, "Creating complex network services with ebpf: Experience and lessons learned"

[6] Autori Linux Kernel, Linux Kernel Documentation, *Azioni XDP*, https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html

[7] Kernel Documentation, *Struttura skb*, http://vger.kernel.org/~davem/skb.html http://vger.kernel.org/ davem/skb.html

[8] Autori Nat-Lab, Github, *XDP-router*, https://github.com/Nat-Lab/xdp-router

[9] NXP Documentation, *LS1046A Freeway Board*, https://www.nxp.com/design/qoriq-developer-resources/ls1046a-freeway-board:FRWY-LS1046A

[10] NXP Documentation, *QMan Introduction*, https://docs.nxp.com/bundle/GUID-C241BB12-95F6-4D6B-A205-7EFD35551DE2/page/GUID-182343BA-04FF-4C53-A2A5-61B9751B6395.html

[11] NXP Documentation, *QMan Scheduling*, https://docs.nxp.com/bundle/GUID-1441E561-3EAD-47FD-A50D-72E1A4E4D69E/page/GUID-9DFCB249-AA7B-41BD-A8F2-F256731C5933.html

[12] NXP Documentation, *QMan: Order Definition/ Restoration*, https://docs.nxp.com/bundle/GUID-1441E561-3EAD-47FD-A50D-72E1A4E4D69E/page/GUID-9DFCB249-AA7B-41BD-A8F2-F256731C5933.html

[13] Autori Linux Kernel, Linux Kernel Documentation, *Libreria libbpf*, https://docs.kernel.org/bpf/libbpf/index.html

[14] Autori Linux Kernel, Linux Kernel Documentation, *Libreria iproute2*, https://git.kernel.org/pub/scm/network/iproute2/iproute2.git

[15] Stephen Hemminger - Linux developer, iproute2 repository, *Header iproute2/bpf.h*, https://github.com/CumulusNetworks/iproute2/blob/master/include/linux/bpf.h

[16] IETF, RFC IETF, *Time To Live*, https://datatracker.ietf.org/doc/html/rfc791#section-1.4

[17] IBM Documentations, *__synch_fetch_and_add*, https://www.ibm.com/docs/en/xl-c-aix/13.1.0?topic=functions-sync-fetch-add

[18] Linux Kernel, *pcpu_copy_value*, https://github.com/torvalds/linux/blob/v5.4/kernel/bpf/hashtab.c#L784

[19] Linux Kernel, *pcpu_init_value*, https://github.com/torvalds/linux/blob/v5.10/kernel/bpf/hashtab.c#L920

[20] Linux Kernel, *BPF_MAP_TYPE_LRU_PERCPU_HASH*, https://docs.kernel.org/bpf/map_hash.html

[21] Systemd official website, *systemd*, http://www.systemd.it

[22] TRex Cisco official website, *TRex Introduction*, https://trex-tgn.cisco.com

[23] IxLoad Keysight official website, *IxLoad Introduction*, https://www.keysight.com/us/en/products/network-test/protocol-load-test/ixload.html

[24] Linux Kernel Documentation, *Flow Offload*, https://www.kernel.org/doc/html/latest/networking/nf_flowtable.html

[25] Linux Manual Page, *mpstat command*, https://man7.org/linux/man-pages/man1/mpstat.1.html