Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Bilanciamento del carico per servizi di accesso ai dati a elevata efficienza utilizzando eXpress Data Path

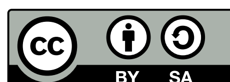
Relatore:
Prof. Renzo Davoli

Correlatore:
Prof. Francesco Giacomini

Presentata da:
Luca Bassi

I Sessione
Anno Accademico 2022/2023

Be curious.
Read widely. Try new things.
I think a lot of what people call intelligence
just boils down to curiosity.
Aaron Swartz



Introduzione

Nel calcolo distribuito è spesso necessario gestire un elevato numero di richieste o task da eseguire. Vengono quindi incrementate le risorse disponibili, per esempio i server, ed è quindi necessario distribuire le risorse e i task tra questi.

Il load balancing è la tecnica che distribuisce il carico di lavoro tra diversi server con lo scopo di rendere più efficiente l'elaborazione, aumentare l'affidabilità e migliorare il tempo di risposta dei servizi. In questo lavoro parliamo di load balancing di traffico di rete. Esistono diverse tipologie di load balancer: hardware o software, statici o dinamici, di livello 4 o 7.

eBPF è una tecnologia che permette di eseguire programmi all'interno di un sandbox del kernel Linux. È particolarmente interessante perché permette di aggiungere funzionalità al kernel senza doverlo ricompilare ed evitando l'uso di moduli. XDP (eXpress Data Path) è un framework di eBPF che permette di processare pacchetti di rete in modo molto efficiente prima che vengano elaborati dallo stack di rete del kernel.

L'obiettivo di questa tesi è mostrare come sia possibile sviluppare un load balancer software utilizzando XDP.

Nel capitolo 1 viene presentata una tassonomia applicabile alle tecniche di load balancing. Nel capitolo 2 vengono illustrate varie soluzioni per il load balancing. Nel capitolo 3 vengono introdotti la tecnologia eBPF e il framework XDP. Nel capitolo 4 sono presentati esempi di programmi XDP di complessità crescente, fino ad arrivare all'implementazione di un load balancer; di questo, nel capitolo 5, vengono mostrati i risultati della valutazione delle prestazioni. Nel capitolo 6 vengono anticipati alcuni promettenti sviluppi futuri, in particolare come distribuire il servizio di load balancing direttamente sui server di backend.

Indice

Introduzione **i**

1 Introduzione al load balancing **1**

1.1 Load balancer: hardware e software . . . 1

1.2 Load balancer: statici e dinamici . . . 2

1.3 Load balancer: livello 4 e livello 7 . . . 2

2 Tecniche e software per il load balancing **5**

2.1 Round-robin DNS . . . 5

2.2 nginx . . . 6

2.3 Linux Virtual Server . . . 6

2.4 ECMP . . . 7

2.5 GitHub Load Balancer Director . . . 8

2.6 Katran . . . 10

2.7 Unimog . . . 12

3 eBPF e XDP **17**

3.1 eBPF . . . 17

3.1.1 Esecuzione event-driven . . . 18

3.1.2 Bytecode . . . 18

3.1.3 Verificatore e sicurezza . . . 18

3.1.4 Compilazione Just-In-Time . . . 19

3.1.5 Mappe . . . 19

3.1.6 Funzioni . . . 20

3.2 XDP 20
3.2.1 Codici di ritorno di XDP 23
3.2.2 Utilizzi di XDP 24
3.2.3 Modalità di funzionamento 26

4 Programmi XDP **27**
4.1 Dipendenze 27
4.2 Primo programma XDP 27
4.3 Mappe 29
4.3.1 Pinning delle mappe 33
4.4 Implementazione di un load balancer con XDP 34
4.4.1 Inoltrare i pacchetti 35
4.4.2 Direct Server Return 40
4.4.3 Selezione del server tramite hashing 43
4.4.4 Impostare i pesi 47

5 Testing **51**
5.1 Ambiente di test 51
5.2 Risultati 55

6 Verso un load balancing distribuito **59**
6.1 Ambiente virtuale 59
6.2 Load balancer XDP distribuito 63

Conclusioni e sviluppi futuri **67**

Bibliografia **69**

Elenco delle figure

2.1 Utilizzo di LVS con Direct Server Return 7
2.2 Schema di funzionamento di GLB Director 9
2.3 Architettura di Katran 11
2.4 Schema di funzionamento di Unimog 14

3.1 Schema dell'elaborazione dei pacchetti tramite XDP 21

6.1 Topologia di rete creata usando GNS3 60

Elenco dei codici sorgente

4.1 Struttura base di un programma XDP 27
4.2 Programma XDP che conteggia il numero di ping 30
4.3 Programma che legge i valori di una mappa dallo user space 33
4.4 Esempio di inoltro di pacchetti tramite XDP 35
4.5 Implementazione del Direct Server Return 41
4.6 Utilizzo dell'hashing per la selezione del server 44
4.7 Programma XDP che legge i pesi da una mappa 47
4.8 Programma per impostare i pesi da user space 50
5.1 Programma XDP usato nei test 52
6.1 Load balancer che utilizza XDP installabile direttamente sui server . . . 63

Capitolo 1

Introduzione al load balancing

Nel calcolo distribuito è spesso necessario gestire un elevato numero di richieste o task da eseguire. Vengono quindi incrementate le risorse disponibili, per esempio il numero di server, ed è quindi necessario distribuire le risorse e i task tra questi.

Il load balancing è la tecnica che distribuisce il carico di lavoro tra vari server. È utilizzata in particolare per distribuire il traffico di rete tra diversi server, per aumentare l'affidabilità e migliorare i tempi di risposta del servizio.

È possibile dividere i load balancer in varie categorie.

1.1 Load balancer: hardware e software

Una prima classificazione può essere effettuata a seconda se è implementato a livello hardware o software.

I load balancer hardware sono dispositivi fisici dedicati che vengono collegati tra il router e i server di backend. Sono dispositivi specializzati e progettati per ottenere le migliori prestazioni. L'integrazione di load balancer hardware potrebbe non essere banale nel caso di architetture complesse e la loro configurazione potrebbe non essere flessibile quanto quella di un load balancer software. Altra caratteristica da tenere in considerazione è l'elevato costo, vista anche la necessità di ridondanza per evitare Single Point of Failure.

I load balancer software sono programmi che permettono di distribuire il traffico tra vari server. Sono installabili su normali macchine preferibilmente dotate di buone schede di rete per evitare che diventino colli di bottiglia. Per quanto sia necessario anche in questo caso avere i servizi ridondati, la possibilità di utilizzare normali server e la presenza di molte soluzioni di questo tipo libere e open source, li rende una soluzione economicamente più accessibile. Sono anche la soluzione più flessibile e personalizzabile. Il load balancing software per reti ad alte prestazioni è fortemente vincolato all'efficienza delle scelte implementative che si fanno.

1.2 Load balancer: statici e dinamici

I load balancer scelgono come distribuire le richieste utilizzando degli algoritmi che possono essere statici o dinamici.

Gli algoritmi statici ignorano lo stato e il carico attuale dei server ed effettuano le scelte basandosi su assunzioni fatte a priori. Per esempio, se un server è il doppio più potente rispetto a un altro possiamo impostare un algoritmo statico per inviare il doppio delle richieste al primo.

Nel caso di algoritmi dinamici invece le richieste vengono inoltrate prendendo in considerazione le condizioni di carico attuali dei server e richiedono pertanto un'infrastruttura parallela di monitoraggio dei sistemi coinvolti. In particolare, non tutte le richieste sono computazionalmente equivalenti, quindi, anche nel caso di server identici, non è detto che dividere in modo equivalente le connessioni tra le varie macchine sia la scelta migliore, cioè che porti a un carico equivalente.

1.3 Load balancer: livello 4 e livello 7

Il modello ISO/OSI [1] è uno standard che definisce l'architettura di reti di calcolatori. È una struttura a pila composta da 7 livelli:

1. Fisico
2. Collegamento

3. Rete

4. Trasporto

5. Sessione

6. Presentazione

7. Applicazione

In particolare i load balancer possono agire a livello 4 (trasporto) o livello 7 (applicazione).

I load balancer di livello 4 agiscono ispezionando i pacchetti fino al livello 4 a cui corrispondono principalmente i protocolli TCP e UDP. Sono normalmente più veloci rispetto ai load balancer di livello 7 perché lavorano a livello più basso, ma per questo sono anche meno configurabili. Per esempio non possono distinguere che tipo di richiesta HTTP è stata effettuata, ma solamente gli indirizzi IP sorgente e destinazione e le porte sorgente e destinazione.

I load balancer di livello 7 invece esaminano i pacchetti analizzando il livello applicazione, per esempio le richieste HTTP. La configurazione avviene quindi a livello più alto ed è per esempio possibile distribuire in modo diverso le richieste per documenti HTML rispetto a quelle per immagini. Lo svantaggio principale di queste soluzioni è che, oltre a essere meno efficienti, supportano di solito solo alcuni protocolli di livello 7, di cui ne esistono una grande varietà, e quindi potrebbero non essere facilmente utilizzabili con protocolli meno diffusi.

Capitolo 2

Tecniche e software per il load balancing

In questo capitolo vengono illustrare le tecniche più diffuse per il load balancing.

2.1 Round-robin DNS

È possibile utilizzare il DNS, cioè il sistema utilizzato per tradurre i nomi di dominio in indirizzi IP, per distribuire il traffico tra diversi server. Si possono infatti assegnare più indirizzi IP allo stesso record DNS permutando la sequenza ogni volta che viene ricevuta una richiesta DNS.

Questa soluzione è molto semplice da configurare e non richiede hardware o software specializzato.

Ha però diverse limitazioni. Infatti le risposte DNS vengono spesso salvate in cache quindi richieste successive ricevono lo stesso IP invece che IP differenti. È inoltre un algoritmo statico che ignora il carico dei server, presuppone che tutte le richieste siano computazionalmente equivalenti e che tutti i server abbiano capacità equivalenti.

2.2 nginx

nginx [2] è un web server open source. Tra le sue innumerevoli funzionalità può essere anche utilizzato come load balancer di livello 7: si imposta per ascoltare sulla porta dove gli utenti si connettono e redireziona il traffico sui server di backend.

Rispetto al round-robin DNS è possibile utilizzare diversi algoritmi per distribuire il traffico: round-robin, least-connected (la successiva richiesta è assegnata al server con meno connessioni attive), ip-hash (viene usata una funzione di hash per determinare il server a cui inoltrare la richiesta). Lavorando a livello 7 può essere più espressivo rispetto ai load balancer di livello 4.

L'implementazione del reverse proxy di nginx supporta il load balancing per HTTP, HTTPS, FastCGI, uwsgi, SCGI, memcached e gRPC. Altri protocolli possono essere supportati solo al costo di modifiche al codice di nginx o dello sviluppo di moduli aggiuntivi. Essendo un load balancer di livello 7, può risultare meno efficiente rispetto a quelli di livello 4.

Esistono anche altri software con funzionalità simili, per esempio HAProxy, Envoy, Traefik, Caddy etc.

2.3 Linux Virtual Server

Un'altra soluzione per il load balancing molto diffusa su Linux è Linux Virtual Server (LVS). LVS viene installato su un server (chiamato *router*) e permette di distribuire il traffico tra vari server di backend (chiamati *real server*). La ridondanza viene gestita da un router di backup che controlla tramite un *heartbeat* se il router attivo funziona correttamente.

Nella configurazione di base tutto il traffico (sia in ingresso, sia in uscita) passa attraverso il server router. Questa circostanza può non essere desiderabile in quanto il server router rischia di fare da collo di bottiglia, oltre a non riuscire in generale a fornire le prestazioni aggregate dei server di backend. LVS permette però di usare il Direct Server Return (vd. sezione 4.4.2), chiamato nel contesto di LVS anche *direct routing*, ovvero sono direttamente i real server a inviare le risposte ai client. In questo modo il

traffico in uscita non passa per il server router diminuendo il carico di quest'ultimo e la possibilità che faccia da collo di bottiglia.

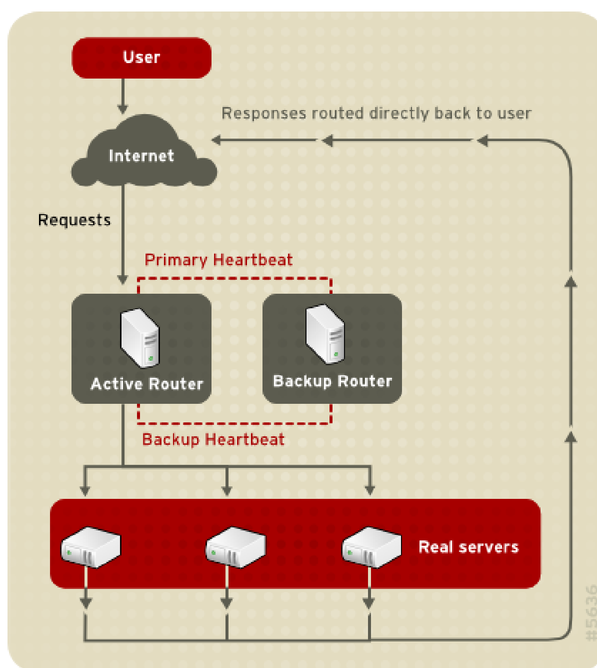


Figura 2.1: Utilizzo di LVS con Direct Server Return (Immagine di Red Hat distribuita con licenza Creative Commons BY-SA 3.0)

LVS permette di usare diversi algoritmi per il load balancing e, tramite il Direct Server Return, evitare che il traffico in uscita passi per il router LVS, attraverso cui però passa tutto il traffico in ingresso. Non impatta quindi sulle prestazioni dei download però può fare da collo di bottiglia nel caso di upload di file.

2.4 ECMP

Il *routing* è il processo di selezione del cammino, attraverso una o più reti, tramite cui trasferire i pacchetti verso un *host*. Possono esistere più percorsi possibili per raggiungere un indirizzo IP. Esistono protocolli di routing dinamici, come per esempio Border Gateway Protocol (BGP) [3], o statici per identificare o annunciare collegamenti, ai quali viene associato un costo.

L'Equal-Cost Multi-Path (ECMP) routing è una strategia di routing, indipendente dal protocollo di routing, pensata per dividere il traffico destinato a un singolo indirizzo IP attraverso collegamenti con lo stesso costo. ECMP calcola l'hash di alcune informazioni del pacchetto in arrivo, per esempio indirizzi IP sorgente e destinazione e porte sorgente e destinazione, e lo utilizza per instradare il pacchetto verso uno dei collegamenti. Si noti come pacchetti successivi di una stessa connessione TCP, avendo lo stesso hash, vengano instradati sempre verso lo stesso collegamento. In questo modo si evita che i pacchetti arrivino in modo disordinato e si mantiene un'affinità di sessione.

Un uso alternativo di ECMP, che lo rende valido anche come soluzione di load balancing, è utilizzarlo per dividere il traffico tra vari server invece che, tramite collegamenti diversi, verso un unico server. Ogni server può annunciare lo stesso indirizzo IP tramite BGP o protocolli simili; in questo modo le connessioni saranno distribuite tra questi server: il router non è a conoscenza che le connessioni non raggiungono la stessa macchina come nel caso classico, ma sono invece gestiste da server diversi.

Quando l'insieme di server che annunciano lo stesso indirizzo IP varia, le connessioni cambiano per ribilanciare il traffico tra i vari server. Questo non è un problema nel caso uno dei server venga spento perché in ogni caso quelle connessioni si interromperebbero. Diventa invece un problema nel caso venga aggiunto un server perché il router, essendo *stateless*, invierà parte dei pacchetti delle connessioni precedentemente presenti al nuovo server, il quale, non sapendo come gestirle (tranne nel caso di pacchetti SYN o di connessioni UDP), le resetterà.

2.5 GitHub Load Balancer Director

GitHub Load Balancer Director [4] è un load balancer di livello 4 sviluppato da GitHub per distribuire il traffico tra vari server minimizzando le interruzioni delle connessioni nel caso di cambiamenti (aggiunte o rimozioni) nell'insieme dei server.

Viene installato su delle macchine dedicate che annunciano lo stesso indirizzo IP e ricevono ognuna una parte dei pacchetti in ingresso tramite ECMP.

Per ogni connessione in arrivo GLB Director seleziona un server primario e uno secondario. Quando il pacchetto arriva al server primario, se non è valido (per esempio

appartiene a una sessione TCP di cui non è presente il corrispondente *socket*), viene inoltrato al server secondario. In questo modo l'aggiunta di un nuovo server non interferisce con le connessioni TCP già iniziate, dando di fatto una seconda possibilità ai pacchetti di arrivare al server corretto.

Per conoscere quali server selezionare come primario e secondario viene generata una tabella di inoltro statica con in ogni riga gli indirizzi IP del server primario e secondario. All'arrivo di un pacchetto viene generato un hash a partire dagli indirizzi IP sorgente e destinazione e dalle porte sorgente e destinazione che viene usato per selezionare la riga della tabella. Di conseguenza, tutti i pacchetti appartenenti alla stessa connessione verranno inoltrati inoltrati sempre allo stesso server.

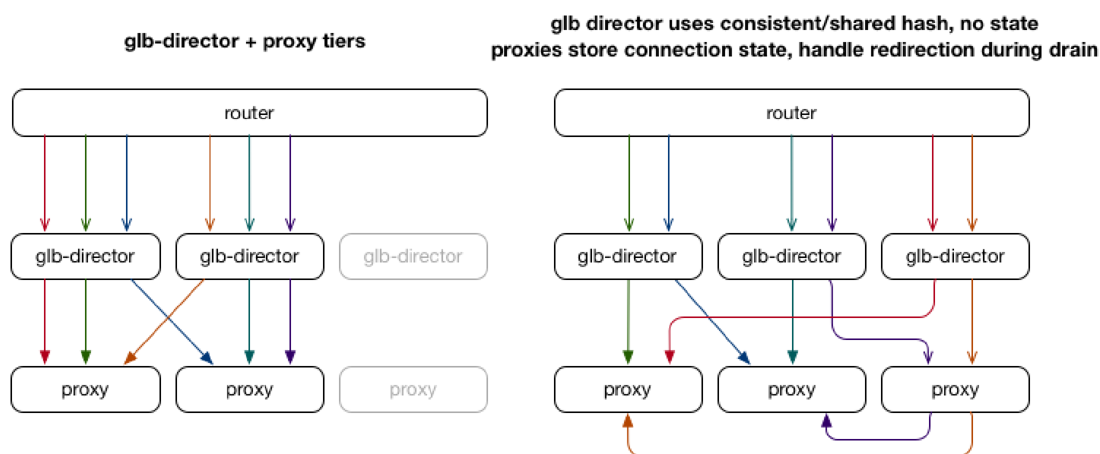


Figura 2.2: Schema di funzionamento di GLB Director (Copyright GitHub)

Gli indirizzi IP dei server devono ovviamente comparire lo stesso numero di volte come server primario e secondario e non devono comparire mai due volte nella stessa riga. Quando viene aggiunto un nuovo server al sistema, alcuni server primari diventano secondari, in modo da permettere al nuovo server di diventare primario in alcune righe; allo stesso tempo diventa secondario in altre. In modo speculare, quando un server viene rimosso dal sistema, nelle righe in cui era primario, il server secondario diventa primario e un altro server viene inserito come secondario.

GitHub ha deciso di utilizzare il *rendezvous hashing* (vd. sezione 4.4.3) per generare la tabella. Dati gli indirizzi IP dei server, di questi viene calcolato l'hash con il numero di riga e ordinati: la riga della tabella sarà quindi composta dai primi due indirizzi IP secondo questo ordine. In questo modo si è sicuri che nella stessa riga non ci sia due volte lo stesso indirizzo IP e che l'ordine relativo, nel caso di aggiunte o rimozioni di indirizzi IP, rimanga lo stesso. Se la funzione di hashing è buona e pseudo-casuale anche l'ordine lo sarà e quindi la distribuzione degli indirizzi IP sarà come richiesta: ogni indirizzo IP compare circa lo stesso numero di volte ed è distribuito tra primario e secondario.

Avendo nella tabella di inoltro solo un server primario e uno secondario è importante gestire correttamente il caso in cui uno dei server venga rimosso. Per limitare al minimo le interruzioni delle connessioni, quando un server deve essere spento, per esempio per manutenzione, nelle righe in cui è presente come server primario viene impostato come secondario, in questo modo non gli verranno inoltrate nuove connessioni, ma continuerà a ricevere quelle già presenti.

Poiché le tabelle di inoltro sono presenti solo nei server GLB Director, è necessario incorporare le informazioni del server secondario nel pacchetto inoltrato al server primario. GitHub, dopo aver testato IP in IP e Foo-over-UDP, ha scelto di incapsulare i pacchetti da inoltrare utilizzando l'header privato del Generic UDP Encapsulation (GUE) [5]: in questo modo i router vedono soltanto normale traffico UDP tra due server.

Quando un server deve inviare la risposta al client lo fa direttamente utilizzando il Direct Server Return, così il traffico in uscita non passa attraverso i server GLB Director.

Per inoltrare i pacchetti, i server GLB Director utilizzano DPDK [6], che permette di processare i pacchetti in modo molto efficiente effettuando il bypass del kernel Linux. La principale limitazione di DPDK è che supporta solo un numero limitato di schede di rete.

2.6 Katran

Nel 2018 Facebook ha reso open source il suo load balancer di livello 4 chiamato Katran [7]. Rispetto alle altre soluzioni viste finora, Katran può essere installato sugli

stessi server in cui sono presenti i servizi, senza quindi richiedere delle macchine separate per il load balancing.

Tutti i server annunciano un IP virtuale tramite ExaBGP [8] e il router distribuisce quindi i pacchetti tra i vari server utilizzando ECMP.

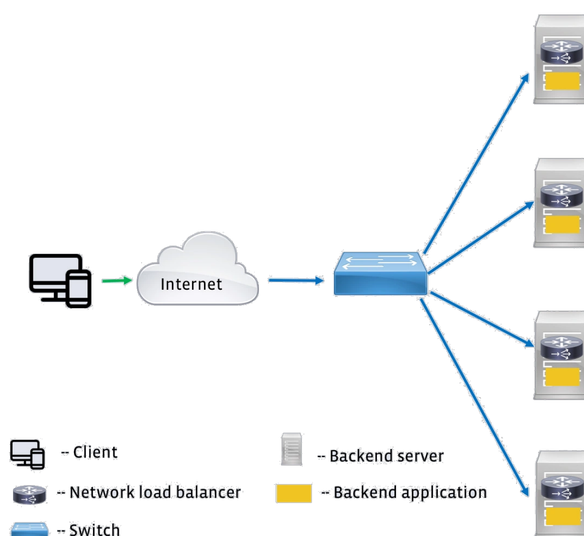


Figura 2.3: Architettura di Katran (Copyright Facebook)

La selezione del server avviene facendo l'hashing, tramite una versione estesa di Maglev [9], degli indirizzi IP sorgente e destinazione, delle porte sorgente e destinazione e del protocollo: in questo modo tutti i pacchetti appartenenti alla stessa connessione vengono inoltrati allo stesso server.

Per inoltrare i pacchetti viene utilizzato eXpress Data Path (XDP) [10] che utilizza la CPU solamente quando viene ricevuto un pacchetto e, al contrario di DPDK, evita il kernel bypass, permettendo di installare il load balancer sulle stesse macchine senza senza un impatto misurabile sulle prestazioni. I pacchetti da inoltrare vengono incapsulati utilizzando IP-in-IP tramite il modulo IPVS del kernel. Viene utilizzato il Direct Server Return, cioè i pacchetti in uscita vengono inviati direttamente al client.

Katran è affiancato da un *control plane* per controllare lo stato dei server e aggiungere o rimuovere IP virtuali.

Katran non supporta pacchetti con opzioni IP e la dimensione massima non può superare 3.5 KB. Non supporta la frammentazione dei pacchetti, né può frammentarli se superano l'MTU (Maximum Transmission Unit) cioè la dimensione massima dei pacchetti.

2.7 Unimog

Unimog [11] è un load balancer di livello 4 sviluppato da Cloudflare. Unimog prende ispirazione da GLB Director, però come Katran viene installato sulle stesse macchine in cui sono presenti i servizi invece che su macchine dedicate al load balancing, trasformando quindi ogni server in un load balancer. Unimog fa load balancing dinamico perché presuppone che non tutti i server abbiano la stessa capacità computazionale, non tutte le richieste siano computazionalmente equivalenti e ci possano essere altri servizi, indipendenti dalle richieste dei client, che utilizzano le risorse dei server.

I server annunciano IP virtuali a cui i client si connettono. Anche nel caso di Unimog viene utilizzato ECMP per inoltrare i pacchetti dal router ai vari server su cui è installato Unimog, che si occupa poi di inoltrarli al server corretto. Non è sufficiente cambiare l'IP di destinazione con l'IP del server a cui si vuole inoltrare i pacchetti, poiché questo nasconderebbe l'indirizzo IP di destinazione originale ed è quindi necessario incapsulare i pacchetti. Unimog usa il Generic UPD Encapsulation (GUE) come GLB Director. A causa dell'incapsulamento i pacchetti diventano più grandi; per esempio un pacchetto da 1500 byte diventa di 1536 byte ed è quindi necessario aumentare l'MTU.

Unimog processa ogni pacchetto in arrivo tramite un programma XDP nel seguente modo:

1. Se il pacchetto non è destinato a un indirizzo IP virtuale lo passa allo stack di rete del kernel; in questo modo il traffico normale non viene modificato.
2. Altrimenti, determina l'indirizzo IP reale a cui deve essere inoltrato il pacchetto.
3. Incapsula il pacchetto e lo inoltra al server corretto.

Al passo 2 tutti i load balancer devono agire in modo coerente nell'inoltrare i pacchetti, indipendentemente da quale programma XDP inoltra il pacchetto. Questo avviene

tramite algoritmi di hashing; nel caso di Unimog viene generato un hash a partire dagli indirizzi IP sorgente e destinazione e dalle porte sorgente e destinazione, che viene utilizzato poi per trovare l'indirizzo IP del server a cui inoltrare il pacchetto. La tabella di inoltro è un array, con una lunghezza che è una potenza di 2, e vengono utilizzati gli N bit meno significativi dell'hash come indice. Essendo la tabella degli inoltri statica, non è un problema se a seguito di una modifica dell'insieme dei server cambia a quale server il router con ECMP inoltra i pacchetti con un determinato hash. L'array è molto più grande del numero di server, così che modificando il numero di volte che ogni server compare è possibile bilanciare il traffico.

Per evitare che un cambio di un valore nell'array interrompa delle connessioni già avviate, viene usata la tecnica del *daisy chaining*. Infatti nel caso di cambiamenti nella tabella di inoltro, alcuni pacchetti inizieranno a essere inoltrati al nuovo server. Se sono pacchetti TCP SYN, cioè che iniziano una nuova connessione non è un problema, ma se sono pacchetti di una connessione avviata prima del cambio, il nuovo server non avrà il corrispondente socket TCP e quindi invierà un pacchetto RST interrompendo la connessione. Per evitare questi problemi è necessario mantenere una cronologia delle modifiche fatte alla tabella di inoltro, in modo che il nuovo server possa inoltrare i pacchetti per cui non ha un socket al server a cui sarebbero stati precedentemente inviati, mantenendo la connessione valida. Per fare ciò Unimog estende la tabella di inoltro in modo che ogni valore non contenga un solo indirizzo IP, ma due: quello del server attuale e quello del server precedente.

Questo richiede che su ogni server sia installato un *redirector*. La logica del redirector è abbastanza semplice:

- Se il pacchetto è un pacchetto SYN, cioè che inizia una nuova connessione, è processato sempre dal primo server.

- Se il pacchetto appartiene a una connessione con il corrispondente socket TCP sul primo server, è processato da questo.

- Altrimenti viene inoltrato al secondo server: si presuppone appartenga a una connessione iniziata da quest'ultimo quando ancora era il primo server.

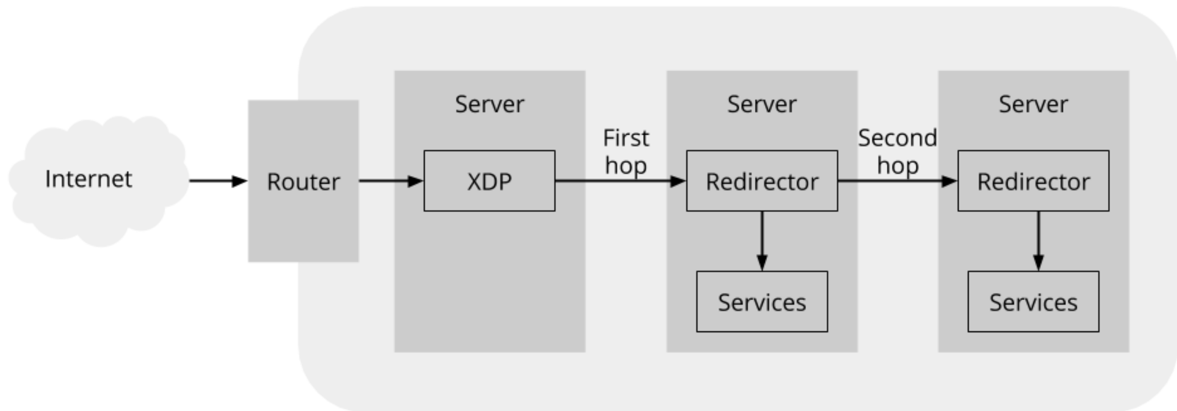


Figura 2.4: Schema di funzionamento di Unimog (Copyright Cloudflare)

Per evitare che il redirector debba accedere un'altra volta alla tabella di inoltro nel caso il pacchetto vada inoltrato al secondo server, l'indirizzo IP di quest'ultimo viene inserito in un *GUE extension header* del pacchetto incapsulato dal programma XDP.

Per quanto il secondo inoltro abbia un costo in termini di prestazioni, Cloudflare riporta che meno dell'1% dei pacchetti viene inoltrato una seconda volta e quindi nella pratica l'impatto è accettabile.

Il redirector inizialmente utilizzato da Cloudflare era quello di GLB, che però era sviluppato da GitHub come un modulo `iptables` per il kernel. I moduli sono più complessi da mantenere rispetto ai programmi eBPF: per esempio è necessario ricompilarli per ogni versione del kernel. Lo hanno quindi implementato da zero come un programma TC classifier (TC [12] è il *traffic control subsystem* nello stack di rete di Linux). Hanno scelto di non implementarlo con XDP poiché è più difficile eseguire il debug di questa tipologia di programmi; per esempio i pacchetti sono visibili da `tcpdump` solo dopo l'elaborazione da parte di XDP.

Unimog memorizza solo l'attuale e il precedente server nella tabella di inoltro. Per evitare di aggiornare in rapida successione due volte la stessa riga della tabella di inoltro e quindi interrompere delle connessioni a causa della cronologia di lunghezza 2, Unimog modifica le righe che sono state meno recentemente aggiornate. Unimog tende anche, quando possibile, a scambiare i valori della stessa riga così da evitare la possibilità che

qualche connessione si interrompa. In questo caso le nuove connessioni andranno al nuovo primo server, che precedentemente era il secondo, bilanciando il carico tra i due.

Per generare la tabella di inoltro il control plane di Unimog prende in considerazione informazioni da diverse fonti:

- Informazioni sui server: Unimog deve sapere quali server sono presenti e i loro indirizzi IP.
- Stato: Unimog deve conoscere lo stato dei server (per esempio non deve inoltrare pacchetti a un server non raggiungibile) e dei servizi installati sui server.
- Carico: Unimog deve conoscere le risorse utilizzate di ogni server.
- Informazioni sugli indirizzi IP: Cloudflare gestisce moltissimi indirizzi IP e lo fa in modo molto dinamico, quindi possono cambiare spesso.

Il control plane è solamente uno in ogni datacenter, ma è presente un'istanza di backup. Viene usato HashiCorp Consul [13] per:

- Propagare ai vari programmi XDP le tabelle di inoltro e le informazioni sugli indirizzi IP virtuali tramite lo store chiave-valore.
- Controllare lo stato dei server e dei servizi.
- Controllare che ci sia solo un'istanza del control plane attiva contemporaneamente tramite dei *lock* distribuiti.

Il control plane ottiene il carico dei server da Prometheus [14] e modifica la tabella di inoltro periodicamente per inviare meno connessioni ai server sovraccarichi e di più a quelli con meno carico. Per determinare se un server è sovraccarico o meno, viene paragonato il carico attuale a quello medio e le modifiche sono proporzionali alla deviazione dalla media. In questo modo il carico di tutti i server converge alla media.

Le informazioni sugli indirizzi IP vengono invece ottenute tramite API interne di Cloudflare.

Il control plane è una componente fondamentale e il loop di feedback è molto complesso poiché reagisce alle variazioni dei server e i server reagiscono alle variazioni che

vengono inviate dal control plane. Per esempio a Cloudflare è capitato che, a seguito di un attacco, molti server venissero contrassegnati come degradati e quindi il control plane ha correttamente iniziato a inviare tutte le connessioni agli altri server, che per questo si sono degradati a loro volta, dando il tempo ai server inizialmente degradati di tornare operativi. A questo punto il control plane ha iniziato a inviare tutte le richieste della seconda metà dei server alla prima e così via anche dopo la fine dell'attacco. È importante quindi distinguere tra un singolo server degradato e molteplici.

Unimog supporta anche il traffico UDP. Nel caso di semplici richieste-risposte, come per il DNS, non è necessario inviare diversi pacchetti allo stesso server perché ce n'è uno solo. Unimog in questo caso distribuisce solo i pacchetti e non è necessario che la tabella di inoltro utilizzi la tecnica del daisy chaining. Alcune applicazioni però potrebbero avere flussi di pacchetti UDP. In questo caso è necessario che tutti i pacchetti vengano inoltrati allo stesso server e quindi viene usata la tecnica del daisy chaining. I pacchetti UDP non hanno pacchetti SYN come nel caso di TCP per segnalare una nuova connessione, quindi la logica di inoltro diventa:

- Se il pacchetto UDP ha un socket corrispondente nel primo server viene processato da questo.
- Altrimenti è inoltrato al secondo server presupponendo che lì ci sia un socket per quel flusso UDP.

Anche se all'apparenza è molto simile alla logica di inoltro dei pacchetti TCP, in questo caso i nuovi flussi vengono gestiti dal secondo server. Quindi quando si vuole aggiungere un nuovo server, questo va aggiunto come primo server nella tabella di inoltro TCP e come secondo in quella UDP. Questo comporta anche che, mentre nel caso di TCP la maggior parte dei pacchetti non viene inoltrato una seconda volta, nel caso di UDP i pacchetti corrispondenti alle nuove connessioni vengono sempre inoltrati due volte.

Capitolo 3

eBPF e XDP

Nel precedente capitolo è stato citato diverse volte XDP. In questo capitolo si definiscono più approfonditamente eBPF e XDP (eXpress Data Path).

3.1 eBPF

Extended BPF [15] è una tecnologia nata come estensione del Berkeley Packet Filter, rinominato per questo classic BPF o cBPF. La tecnologia si è espansa oltre la parte di networking originale e per questo eBPF non viene più usato come acronimo e a volte viene chiamato semplicemente BPF.

eBPF permette di eseguire programmi in un sandbox all'interno di un contesto privilegiato come quello del kernel. Questo permette di espandere le funzionalità del kernel senza doverlo ricompilare né utilizzare i moduli. I principali campi di applicazione di eBPF sono il networking, la sicurezza, l'osservabilità e la profilazione. Il modello di esecuzione di eBPF è principalmente *event-driven*. Le prestazioni dei programmi eBPF sono paragonabili a quelli nativi grazie all'implementazione nel kernel di un compilatore Just-In-Time (vd. sezione 3.1.4). Essendo programmi che agiscono a livello kernel, prima dell'esecuzione vengono verificati con l'analisi statica per evitare, per esempio, che leggano aree di memoria protette oppure eseguano loop infiniti.

3.1.1 Esecuzione event-driven

Come detto in precedenza i programmi eBPF sono event-driven, cioè vengono eseguiti quando il kernel o un'applicazione passano un certo punto di *hook*. Hook predefiniti includono *system call*, entrata e uscita di funzioni, *tracepoint* del kernel, eventi di rete, etc. È anche possibile creare un *kernel probe* (`kprobe`) o uno *user probe* (`uprobe`) per collegare un programma eBPF praticamente ovunque.

3.1.2 Bytecode

I programmi eBPF vengono caricati nel kernel Linux sotto forma di *bytecode* [16], un linguaggio intermedio specifico ottenuto dalla compilazione di un programma. Clang [17] permette di compilare programmi C in bytecode. Spesso vengono usate librerie di alto livello per scrivere i programmi eBPF. Una volta identificato l'hook desiderato, il programma eBPF viene caricato tramite la system call `bpf` oppure tramite le funzioni delle librerie eBPF.

3.1.3 Verificatore e sicurezza

Dopo aver chiamato la system call `bpf`, il kernel Linux verifica che sia sicuro eseguire il bytecode caricato. Viene controllato che il processo che carica il programma eBPF nel kernel Linux abbia i privilegi richiesti: deve essere eseguito come root o richiedere la *capability* `CAP_BPF`. È possibile attivare una versione ridotta di eBPF che non richiede privilegi, ma ha meno funzionalità e accesso limitato al kernel.

Il verificatore convalida che il programma eBPF non contenga loop infiniti, non si blocchi e l'esecuzione termini sempre. Si possono usare loop, ma il programma è accettato solo se il verificatore riesce a controllare che il loop contiene una condizione di uscita che è garantita diventare vera. Il verificatore controlla che il programma non usi variabili non inizializzate e che non acceda ad aree di memoria protette.

I programmi eBPF hanno un limite di complessità [18], cioè il numero massimo di istruzioni che il verificatore può controllare prima di rifiutare il programma. Il verificatore costruisce infatti un grafo aciclico diretto (DAG) e valuta tutti i possibili percorsi di

esecuzione. Il limite è variato diverse volte e dalla versione 5.2 di Linux è di un milione di istruzioni.

Dopo la verifica viene eseguito un processo di *hardening*. Per esempio la memoria del kernel contenente il programma eBPF viene resa di sola lettura. Se a causa di un software malevolo o un bug del kernel, si prova a modificare il programma eBPF, il kernel andrà in crash impedendo l'esecuzione del codice malevolo. Vengono applicate mitigazioni per gli attacchi speculativi sulle CPU, come Spectre [19]. Viene utilizzato il *constant blinding* per evitare che attaccanti inseriscano codice eseguibile sotto forma di costanti, che in caso di bug del kernel possono permettere di saltare nelle sezioni di memoria del programma eBPF per eseguire codice.

I programmi eBPF non possono accedere direttamente alla memoria del kernel. L'accesso alle strutture dati fuori dal contesto del programma deve essere eseguito tramite funzioni *helper*. Questo garantisce coerenza nell'accesso dei dati rendendolo soggetto ai privilegi del programma eBPF.

3.1.4 Compilazione Just-In-Time

La compilazione Just-In-Time traduce il bytecode generico del programma eBPF nel set di istruzioni specifiche dell'architettura su cui viene eseguito, in modo da ottimizzare le prestazioni. Questo permette ai programmi eBPF di essere efficienti quanto il codice, compilato nativamente, del kernel o dei moduli.

3.1.5 Mappe

Le mappe (in gergo *maps*) sono il metodo usato dai programmi eBPF per salvare e recuperare dati. I programmi eBPF hanno accesso alle mappe a cui anche i programmi nello *user space* possono accedere tramite system call. Le mappe possono essere di diversi tipi, per esempio array o hash table, e possono essere sia condivise sia separate su ogni CPU (vd. sezione 4.3).

3.1.6 Funzioni

I programmi eBPF non possono chiamare tutte le funzioni del kernel, perché questo li renderebbe eseguibili solo su versioni specifiche del kernel e complicherebbe la compatibilità. Utilizzano invece delle funzioni helper in modo che le API siano stabili. Ad esempio, esistono funzioni helper per generare numeri casuali, ottenere data e ora, gestire le mappe e manipolare e inoltrare i pacchetti di rete. Oltre alle chiamate di funzione, è possibile anche eseguire un altro programma eBPF e sostituire il contesto di esecuzione, similmente a quanto fa la system call `execve`.

3.2 XDP

Esistono diversi tipi di programmi eBPF. Un tipo di programma per il networking è eXpress Data Path. XDP è un framework per eBPF che permette di processare i pacchetti di rete in modo efficiente. Esegue il programma eBPF il prima possibile, cioè nel momento in cui il driver di rete riceve il pacchetto. A questo punto il driver prende il pacchetto dai buffer circolari dell'interfaccia di rete (*receive rings*) senza fare nessuna operazione computazionalmente costosa e il programma XDP è eseguito appena il pacchetto diventa disponibile alla CPU per essere processato. XDP lavora di concerto con il kernel Linux e la sua infrastruttura, infatti il kernel Linux non viene bypassato come nei vari framework di rete che operano in user space.

Mantenere il pacchetto nel *kernel space* ha molti vantaggi:

- XDP può riutilizzare nelle funzioni helper di eBPF tutti i driver di reti del kernel, gli strumenti dello user space e anche tutte le infrastrutture disponibili nel kernel come le tabelle di routing, i socket, etc.

- XDP ha lo stesso modello di sicurezza del resto del kernel per accedere all'hardware.

- Non c'è bisogno di passare da kernel space a user space, poiché il pacchetto è già presente nel kernel e può quindi inoltrare i pacchetti ad altre entità come *namespace* usati dai container o lo stack di rete del kernel.

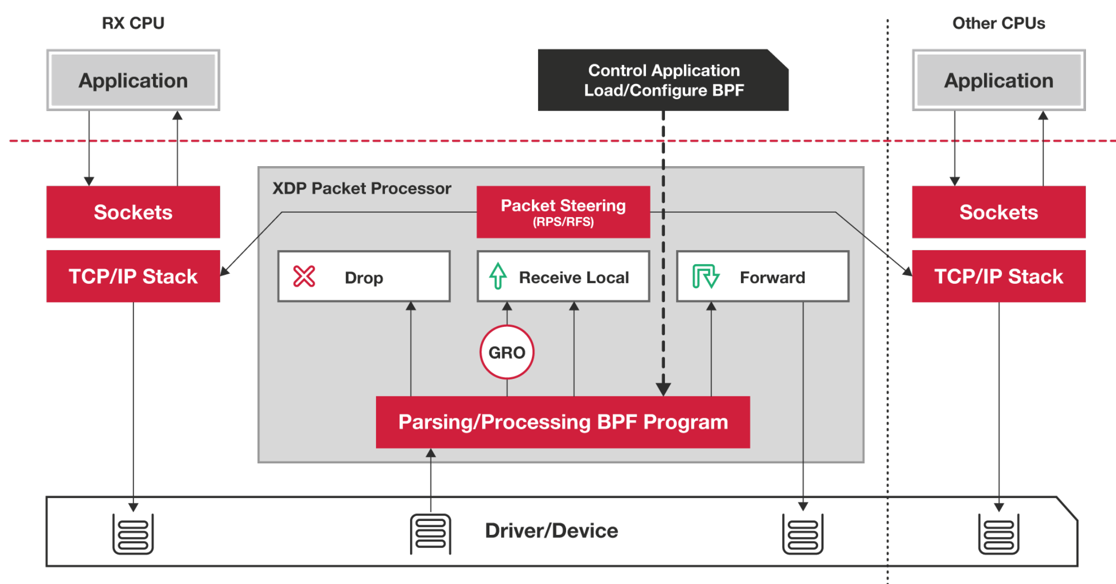


Figura 3.1: Schema dell'elaborazione dei pacchetti tramite XDP (Copyright IO Visor)

- È possibile inoltrare i pacchetti da XDP allo stack TCP/IP del kernel, che è robusto, efficiente e molto usato, invece di usare uno stack TCP/IP separato come nel caso dei framework user space.

- L'utilizzo di eBPF permette una programmabilità completa, mantenendo delle interfacce al sistema operativo (*Application Binary Interface*) stabili con le stesse garanzie di non rompere mai lo user space delle ABI delle system call del kernel. Rispetto ai moduli fornisce anche maggiore sicurezza grazie al verificatore eBPF il quale assicura la stabilità del funzionamento del kernel.

- XDP permette di scambiare atomicamente programmi durante il runtime senza interruzioni del traffico di rete o riavvii del kernel o del sistema.

- XDP permette una strutturazione flessibile dei carichi di lavoro integrati nel kernel. Per esempio può operare sia in modalità *busy polling* sia in modalità *interrupt driven*. Non è richiesto dedicare esplicitamente CPU a XDP. Non ci sono requisiti hardware particolari e non utilizza le *hugepages*.

- XDP non richiede moduli kernel di terze parti o licenze. È una soluzione architetturale di lungo termine, una parte del kernel Linux e sviluppata dalla sua comunità.
- XDP è già attivo e disponibile sulle distribuzioni che utilizzano Linux 4.8+ e supporta la maggior parte dei driver di rete 10 Gigabit o superiori.

XDP si assicura inoltre che i pacchetti siano disposti linearmente e che entrino in una pagina DMA (*Direct Memory Access*) che è leggibile e scrivibile dal programma eBPF. XDP si assicura anche che ci siano 256 byte aggiuntivi disponibili al programma per implementare header d'incapsulamento personalizzati usando la funzione helper di eBPF `bpf_xdp_adjust_head()` o aggiungere metadati aggiuntivi all'inizio del pacchetto tramite `bpf_xdp_adjust_meta()`.

Il framework contiene anche degli *action code* XDP che il programma eBPF ritorna per comunicare al driver cosa fare con il pacchetto e permette di rimpiazzare atomicamente i programmi eBPF nel layer XDP. XDP è stato progetto per avere alte prestazioni. eBPF permette di accedere ai dati dei pacchetti tramite *direct packet access*; questo significa che il programma mantiene i puntatori ai dati direttamente nei registri, carica il contenuto dai registri e li scrive da questi nel pacchetto.

La rappresentazione dei pacchetti in XDP che è passata al programma eBPF con un *BPF context* è la seguente:

```
1  struct xdp_buff {
2      void *data;
3      void *data_end;
4      void *data_meta;
5      void *data_hard_start;
6      struct xdp_rxq_info *rxq;
7      struct xdp_txq_info *txq;
8      u32 frame_sz;
9      u32 flags;
10 };
```

`data` punta all'inizio del pacchetto, `data_end` alla fine. Siccome XDP permette di aggiungere header, `data_hard_start` punta allo spazio massimo dall'inizio della pagina che non si può superare, ciò significa che quando un pacchetto viene incapsulato, `data` viene

spostato verso il limite indicato da data_hard_start tramite bpf_xdp_adjust_head(). La stessa funzione permette l'operazione opposta sui pacchetti. data_meta inizialmente punta alla stessa locazione di memoria di data, ma bpf_xdp_adjust_meta() permette di spostarlo verso data_hard_start per fare spazio a metadati personalizzati che sono invisibili al normale stack di rete del kernel, ma possono essere letti dai programmi eBPF *traffic control* poiché sono trasferiti da XDP a skb, cioè il socket buffer (la struttura dati utilizzata dallo stack di rete del kernel Linux). Viceversa, la stessa funzione può essere usata per rimuovere o ridurre i metadati personalizzati. rxp punta a una struttura che contiene metadati: il programma eBPF può per esempio ottenere il queue_index o informazioni sul dispositivo come l'ifindex (l'indice dell'interfaccia di rete).

```
1  struct xdp_rxq_info {
2      struct net_device *dev;
3      u32 queue_index;
4      u32 reg_state;
5      struct xdp_mem_info mem;
6      unsigned int napi_id;
7      u32 frag_size;
8  };
9  struct xdp_txq_info {
10     struct net_device *dev;
11 };
```

3.2.1 Codici di ritorno di XDP

Il programma XDP al termine deve ritornare un codice per indicare al driver come processare il pacchetto. I codici sono presenti nell'enum xdp_action dell'header file linux/bpf.h.

XDP_ABORTED Il pacchetto viene ignorato (vd. XDP_DROP), ma è possibile monitorare questo codice di ritorno tramite il tracepoint trace_xdp_exception per rilevare comportamenti anomali.

XDP_DROP Il pacchetto deve essere ignorato dal driver senza sprecare ulteriori risorse. Utile per esempio per implementare meccanismi anti-DDoS o firewall.

XDP_PASS Il pacchetto deve essere processato dallo stack di rete del kernel. La CPU che stava elaborando il pacchetto allocherà un `skb` e lo popolerà per passarlo poi all'engine GRO (*Generic Receive Offload*). Questo è quello che succederebbe sempre se il programma XDP non fosse presente.

XDP_TX Il pacchetto viene inoltrato nuovamente dalla stessa scheda di rete su cui era stato ricevuto.

XDP_REDIRECT Simile a `XDP_TX`, ma l'inoltro avviene tramite una diversa scheda di rete selezionabile utilizzando la funzione `bpf_redirect()`.

3.2.2 Utilizzi di XDP

XDP può essere utilizzato in vari modi e viene in particolare sfruttato quando è necessario processare pacchetti in modo performante come nel caso di reti ad alte prestazioni.

Mitigazioni DDoS e firewall

È possibile utilizzare il codice di ritorno `XDP_DROP` per applicare *policy* di rete in modo molto efficiente e con un basso costo computazionale per pacchetto. Questo può essere utilizzato per mitigare attacchi DDoS o implementare regole del firewall a costo quasi zero.

Inoltro e load balancing

Un altro utilizzo rilevante di XDP è quello di inoltrare pacchetti e fare load balancing tramite i codici di ritorno `XDP_TX` e `XDP_REDIRECT`. Il pacchetto può essere modificato dal programma XDP, per esempio incapsulando il pacchetto prima di inviarlo nuovamente. Con `XDP_TX` è possibile implementare load balancer che inoltrano i pacchetti dalla stessa scheda di rete, mentre con `XDP_REDIRECT` possono essere inoltrati da una scheda di rete differente.

Filtraggio e processamento prima dello stack di rete

XDP può anche essere usato per migliorare la sicurezza dello stack di rete del kernel tramite `XDP_DROP`. Infatti è possibile filtrare i pacchetti irrilevanti prima che lo stack di rete li elabori. Si possono per esempio filtrare i pacchetti a seconda che siano traffico TCP, UDP etc. Questo ha il vantaggio che il pacchetto non deve attraversare le varie componenti dello stack di rete come l'engine GRO, il *flow dissector* del kernel etc. prima che sia possibile determinare se ignorarlo, riducendo la superficie di attacco. Grazie al fatto che il programma XDP viene eseguito all'inizio del processamento del pacchetto è possibile considerare trascurabile la penalità introdotta. Inoltre nel caso venisse scoperto un bug nello stack di rete del kernel, XDP potrebbe essere usato per ignorare quella tipologia di pacchetti e mitigare così l'attacco senza dover riavviare il kernel o i servizi. Grazie alla possibilità di sostituire i programma XDP atomicamente è possibile aggiungere e cambiare queste regole senza interrompere il traffico di rete.

Un altro modo di utilizzare il processamento prima dello stack di rete è che, dato che il kernel non ha ancora allocato un `skb` per il pacchetto, il programma XDP lo può modificare e la modifica risulterebbe trasparente allo stack di rete del kernel. Questo può essere utile nel caso si usino protocolli di incapsulamento personalizzati così che il pacchetto possa essere decapsulato prima di essere passato all'engine GRO che altrimenti non saprebbe come gestire il protocollo personalizzato. XDP permette anche di inserire metadati all'inizio del pacchetto. Questi sono invisibili allo stack di rete del kernel, possono essere aggregati tramite GRO e poi processati tramite un programma eBPF traffic control che ha a disposizione il contesto di un `skb`.

Campionamento e monitoraggio dei flussi

XDP può essere usato anche per il campionamento, il monitoraggio e l'analisi dei pacchetti. Per analisi complesse, XDP può anche mettere i pacchetti e i metadati in un *memory mapped ring buffer* dell'infrastruttura `perf` di Linux per un'applicazione nello user space. Questo permette che solo i dati iniziali di un flusso siano analizzati e una volta determinato che è traffico non malevolo, ignorare il monitoraggio. Grazie alla flessibilità di eBPF è quindi possibile implementare campionamenti e monitoraggi personalizzati.

3.2.3 Modalità di funzionamento

XDP ha tre modalità di funzionamento.

Native XDP Il programma eBPF XDP è eseguito direttamente dal driver di rete. Le schede di rete più diffuse supportano questa modalità. Questa è la modalità predefinita.

Offloaded XDP Il programma XDP è eseguito direttamente dalla scheda di rete invece che dal processore dell'host. Quindi il costo computazionale per ogni pacchetto già basso viene spostato dalla CPU alla scheda di rete garantendo prestazioni ancora più alte rispetto alla modalità Native XDP. Questa modalità è implementata da schede di rete che hanno processori *multicore* e *multithread* in cui un compilatore Just-In-Time all'interno del loro kernel traduce il bytecode eBPF in istruzioni native per la scheda di rete. I driver che supportano la modalità Offloaded XDP normalmente supportano anche quella Native XDP nel caso in cui alcune funzioni helper di eBPF non sono ancora disponibili o sono disponibili solo per la modalità Native XDP.

Generic XDP Per i driver che non implementano le due precedenti modalità, il kernel permette di utilizzare questa modalità che non richiede modifiche ai driver poiché viene eseguita successivamente nello stack di rete. Questa è pensata per gli sviluppatori che vogliono scrivere e testare programmi eBPF XDP e non raggiunge le prestazioni delle due precedenti modalità che sono infatti consigliate in ambienti di produzione.

Capitolo 4

Programmi XDP

In questo capitolo si illustrano degli esempi di programmi XDP.

4.1 Dipendenze

Per sviluppare i programmi XDP verrà usato `libbpf` [20]. Saranno necessari anche `llvm`, `clang` e `libelf` per compilare il programma nel bytecode di eBPF e salvarlo in un file ELF. Verranno usati gli `xdp-tools` e in particolare il comando `xdp-loader` per collegare il programma XDP all'interfaccia di rete. Sono necessari i file header del kernel Linux. Queste dipendenze dovrebbero essere facilmente installabili tramite il package manager della distribuzione Linux.

È possibile creare ambienti di test per i programmi XDP tramite gli script presenti nel *repository* di `xdp-tutorial` [21].

4.2 Primo programma XDP

Il seguente programma XDP fa semplicemente passare tutti i pacchetti e quindi non interferisce con la ricezione del traffico di rete.

Codice sorgente 4.1: Struttura base di un programma XDP

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf_helpers.h>
```

```
3
4 SEC("xdp")
5 int xdp_pass(struct xdp_md *ctx)
6 {
7     return XDP_PASS;
8 }
```

SEC("xdp") è una macro definita in bpf_helper.h di libbpf che serve per porre le diverse parti del programma nelle varie sezioni del file elf_bpf. Lo struct xdp_md viene rimappato nello struct xdp_buff dal kernel quando viene caricato il programma.

```
1 struct xdp_md {
2     __u32 data;
3     __u32 data_end;
4     __u32 data_meta;
5     __u32 ingress_ifindex; /* rxq->dev->ifindex */
6     __u32 rx_queue_index;  /* rxq->queue_index  */
7     __u32 egress_ifindex;  /* txq->dev->ifindex */
8 };
```

I codici di ritorno, per esempio XDP_PASS, sono definiti nell'header file linux/bpf.h del kernel Linux. Il nome della funzione sarà anche il nome del programma XDP da collegare all'interfaccia.

È possibile compilare il programma con clang specificando come target bpf:

```
clang -O2 -g -Wall -Wno-compare-distinct-pointer-types \
    -target bpf -c xdp_pass.c -o xdp_pass.o
```

Per verificare che il nome della funzione è anche il nome della sezione nel file ELF si può utilizzare llvm-objdump.

```
$ llvm-objdump -S --no-show-raw-insn xdp_pass.o
xdp_pass.o: file format elf64-bpf
Disassembly of section xdp:
0000000000000000 <xdp_pass>:
;  return XDP_PASS;
       0: r0 = 0x2
       1: exit
```

Lo script `testenv.sh` di `xdp-tutorial` permette di creare un ambiente di test:

```
sudo ./testenv.sh setup --name test
```

A questo punto si può caricare il programma XDP nell'interfaccia di test usando il comando `xdp-loader` presente negli `xdp-tools`:

```
sudo xdp-loader load test xdp_pass.o
```

Per verificare che il programma XDP sia stato collegato all'interfaccia di rete:

```
$ ip link show dev test
6: test@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp [...]
    link/ether 62:d9:a4:33:8a:5b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    prog/xdp id 123
```

Provando a inviare dei ping all'interfaccia di rete, si può notare che viene ricevuta una risposta, cosa attesa visto che il programma XDP fa passare i pacchetti:

```
sudo ./testenv.sh ping --name test
```

Come ulteriore esperimento si può cambiare il codice di ritorno del programma XDP da `XDP_PASS` a `XDP_DROP`, ricompilare e riconnetterlo all'interfaccia di rete. Riprovando a fare dei ping si noterà che non verranno più ricevute risposte.

È possibile eliminare l'ambiente di test con:

```
sudo ./testenv.sh teardown --name test
```

4.3 Mappe

Il seguente programma XDP conta quanti pacchetti ICMP vengono ricevuti tramite IPv4 e quanti tramite IPv6. Per esporre questi dati nello user space vengono utilizzate le mappe. In questo caso la mappa sarà un array di `long` di lunghezza 2. Il programma effettua il *parsing* dell'header del frame Ethernet. A seconda della versione del protocollo IP, fa il parsing dell'header dei pacchetti IP. Incrementa poi il contatore relativo alla versione di IP nella mappa se il protocollo è ICMP.

Codice sorgente 4.2: Programma XDP che conteggia il numero di ping

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <linux/ipv6.h>
6  #include <bpf/bpf_endian.h>
7  #include <bpf/bpf_helpers.h>
8
9  struct {
10     __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
11     __uint(max_entries, 2);
12     __type(key, __u32);
13     __type(value, long);
14     //__uint(pinning, LIBBPF_PIN_BY_NAME);
15 } count SEC(".maps");
16
17 SEC("xdp")
18 int xdp_count(struct xdp_md *ctx)
19 {
20     void *data_end = (void *)(long)ctx->data_end;
21     void *pos = (void *)(long)ctx->data;
22     struct ethhdr *eth = pos;
23     if (eth + 1 > data_end) {
24         return -1;
25     }
26     __u16 h_proto = eth->h_proto;
27     pos = eth + 1;
28     __u32 key;
29     if (h_proto == bpf_htons(ETH_P_IP)) {
30         struct iphdr *ip = pos;
31         if (ip + 1 > data_end) {
32             return -1;
33         }
34         __u8 protocol = ip->protocol;
35         if (protocol == IPPROTO_ICMP) {
36             key = 0;
37             long *value = bpf_map_lookup_elem(&count, &key);
```

```
38              if (value) {
39                  *value += 1;
40              }
41          }
42      } else if (h_proto == bpf_htons(ETH_P_IPV6)) {
43          struct ipv6hdr *ipv6 = pos;
44          if (ipv6 + 1 > data_end) {
45              return -1;
46          }
47          __u8 nexthdr = ipv6->nexthdr;
48          if (nexthdr == IPPROTO_ICMPV6) {
49              key = 1;
50              long *value = bpf_map_lookup_elem(&count, &key);
51              if (value) {
52                  *value += 1;
53              }
54          }
55      }
56      return XDP_PASS;
57  }
```

La mappa è di tipo `BPF_MAP_TYPE_PERCPU_ARRAY` per evitare di dover utilizzare operazioni atomiche necessarie per evitare che le varie CPU vadano a sovrascrivere a vicenda i dati. Poiché i dati dei pacchetti sono acceduti con letture dirette della memoria, il verificatore richiedere che ci siano dei controlli ogni volta che si ha un *offset*. Per questi controlli viene usato il puntatore `data_end` che viene impostato per puntare alla fine del pacchetto durante l'esecuzione. Il puntatore `pos` viene usato per tenere traccia della posizione durante il parsing. Lo `struct ethhdr`, definito in `linux/if_ether.h`, contiene la struttura dell'header del frame Ethernet. Similmente le strutture dati per il parsing degli header dei pacchetti IP. La funzione `bpf_htons`, definita in `bpf/bpf_endian.h`, serve per cambiare l'ordine dei byte da quello dell'host a quello di rete. La funzione `bpf_map_lookup_elem` permette di accedere alle mappe. I protocolli (es. `IPPROTO_ICMP`) sono definiti in `linux/in.h`.

Per provare questo programma XDP è necessario creare un ambiente di test con il supporto a IPv4:

```
sudo ./testenv.sh setup --name test --legacy-ip
```

È possibile fare il ping tramite IPv4 con:

```
sudo ./testenv.sh ping --name test --legacy-ip
```

Si può visualizzare il contenuto delle mappe tramite `bpftool`. Per prima cosa va trovato l'id della mappa (in questo caso è 41):

```
$ sudo bpftool map show | grep count
41: percpu_array  name count  flags 0x0
```

Per visualizzare il contenuto della mappa (da notare che i valori sono divisi tra le varie CPU):

```
$ sudo bpftool map dump id 41
[{
        "key": 0,
        "values": [{
                "cpu": 0,
                "value": 3
            },{
                "cpu": 1,
                "value": 2
            }
        ]
    },{
        "key": 1,
        "values": [{
                "cpu": 0,
                "value": 1
            },{
                "cpu": 1,
                "value": 5
```

```
            }
        ]
    }
]
```

4.3.1 Pinning delle mappe

Nel precedente esempio i valori delle mappe venivano letti tramite `bpftool`. Si può accedere ai valori della mappa da un programma in user space così da poter fare la somma. Il meccanismo di condivisione delle mappe si chiama *pinning*.

Per prima cosa è necessario montare un *file system* di tipo `bpffs` tramite:

```
mount -t bpf bpf /sys/fs/bpf/
```

L'unica modifica da fare al programma XDP precedente è aggiungere allo `struct` della mappa:

```
__uint(pinning, LIBBPF_PIN_BY_NAME);
```

Quando si carica il nuovo programma XDP bisogna specificare il percorso dove salvare le mappe:

```
sudo xdp-loader load test xdp_count.o --pin-path /sys/fs/bpf/test
```

Per accedere alle mappe, prima va letto l'oggetto BPF con la funzione `bpf_obj_get` poi si può usare la funzione `bpf_map_lookup_elem` per accedere ai valori. Questo è il programma che calcola il numero di ping ricevuti tramite IPv4 e IPv6:

Codice sorgente 4.3: Programma che legge i valori di una mappa dallo user space

```
1 #include <linux/bpf.h>
2 #include <bpf/bpf.h>
3 #include <bpf/libbpf.h>
4 #include <stdio.h>
5
6 int main(int argc, char *argv[])
7 {
```

```
8       int fd = bpf_obj_get("/sys/fs/bpf/test/count");
9       if (fd < 0) {
10          printf("Error bpf_obj_get\n");
11          return fd;
12      }
13      int nr_cpus = libbpf_num_possible_cpus();
14      long count_sum[] = { 0, 0 };
15      for (__u32 key = 0; key <= 1; ++key) {
16          long values[nr_cpus];
17          if ((bpf_map_lookup_elem(fd, &key, values)) != 0) {
18              printf("Error bpf_map_lookup_elem\n");
19              return -1;
20          }
21          for (int i = 0; i < nr_cpus; ++i) {
22              count_sum[key] += values[i];
23          }
24      }
25      printf("IPv4: %d\nIPv6: %d\n", count_sum[0], count_sum[1]);
26      return 0;
27  }
```

Si può compilare questo programma con:

```
gcc xdp_count_sum.c -o xdp_count_sum -lbpf
```

Dopo aver fatto dei ping sia tramite IPv4 sia tramite IPv6, è possibile visualizzare il totale con:

```
sudo ./xdp_count_sum
```

4.4 Implementazione di un load balancer con XDP

In questa sezione è illustrato come implementare un load balancer tramite XDP.

4.4.1 Inoltrare i pacchetti

Come primo esempio viene mostrato come sia possibile inoltrare i pacchetti utilizzando il codice di ritorno `XDP_TX` e sfruttarlo per creare un semplice load balancer round-robin.

Per prima cosa è necessario creare un ambiente di test. Verranno utilizzati dei container con l'immagine `nginxdemos/hello:plain-text` che espone un server web nginx alla porta 80 che risponde con delle informazioni sul server.

```
for i in {1..4}; do
    sudo podman run -d --rm \
    --ip "10.88.0.1$i" \
    --mac-address "02:42:00:00:00:1$i" \
    --name "server$i" -h "server$i" \
    nginxdemos/hello:plain-text;
done
```

È possibile testare la connessione a uno dei server con:

```
$ curl 10.88.0.11:80
Server address: 10.88.0.11:80
Server name: server1
Date: 01/Jul/2023:12:34:56 +0000
URI: /
Request ID: 4dbfe4c43374b650d6be6ebf9de3313f
```

Questo programma XDP inoltra le richieste provenienti da un client a uno dei server e viceversa.

Codice sorgente 4.4: Esempio di inoltro di pacchetti tramite XDP

```
1 #include <linux/bpf.h>
2 #include <linux/if_ether.h>
3 #include <linux/in.h>
4 #include <linux/ip.h>
5 #include <bpf/bpf_endian.h>
```

```
6  #include <bpf/bpf_helpers.h>
7
8  #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
9
10 static __always_inline __u16 csum_fold_helper(__u64 csum)
11 {
12     int i;
13 #pragma unroll
14     for (i = 0; i < 4; i++) {
15         if (csum >> 16) {
16             csum = (csum & 0xffff) + (csum >> 16);
17         }
18     }
19     return ~csum;
20 }
21
22 static __always_inline __u16 iph_csum(struct iphdr *iph)
23 {
24     iph->check = 0;
25     unsigned long long csum = bpf_csum_diff(0, 0, (unsigned int *)iph,
26                               sizeof(struct iphdr), 0);
27     return csum_fold_helper(csum);
28 }
29
30 unsigned char client_mac[ETH_ALEN] = {
31     0x02, 0x42, 0x00, 0x00, 0x00, 0x09
32 };
33
34 unsigned char load_balancer_mac[ETH_ALEN] = {
35     0x02, 0x42, 0x00, 0x00, 0x00, 0x10
36 };
37
38 unsigned char server_mac[][ETH_ALEN] = {
39     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x11 },
40     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x12 },
41     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x13 },
42     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x14 },
43 };
```

```
44
45 __be32 client_ip = IP_ADDRESS(10, 88, 0, 9);
46
47 __be32 load_balancer_ip = IP_ADDRESS(10, 88, 0, 10);
48
49 __be32 server_ip[] = {
50     IP_ADDRESS(10, 88, 0, 11),
51     IP_ADDRESS(10, 88, 0, 12),
52     IP_ADDRESS(10, 88, 0, 13),
53     IP_ADDRESS(10, 88, 0, 14),
54 };
55
56 unsigned int backend_server_index = 0;
57
58 SEC("xdp")
59 int xdp_redirect(struct xdp_md *ctx)
60 {
61     void *data_end = (void *)(long)ctx->data_end;
62     void *pos = (void *)(long)ctx->data;
63     struct ethhdr *eth = pos;
64     if (eth + 1 > data_end) {
65         return -1;
66     }
67     __u16 h_proto = eth->h_proto;
68     pos = eth + 1;
69     if (h_proto == bpf_htons(ETH_P_IP)) {
70         struct iphdr *ip = pos;
71         if (ip + 1 > data_end) {
72             return -1;
73         }
74         if (ip->protocol == IPPROTO_TCP ||
75             ip->protocol == IPPROTO_UDP) {
76             if (ip->saddr == client_ip) {
77                 bpf_printk("Sending request to server%d",
78                        backend_server_index + 1);
79                 if (backend_server_index < 4) {
80                     __builtin_memcpy(
81                         eth->h_dest,
```

```
82                              server_mac[backend_server_index],
83                              sizeof(server_mac
84                                          [backend_server_index]));
85                          ip->daddr =
86                              server_ip[backend_server_index];
87                      }
88                      backend_server_index =
89                          (backend_server_index + 1) %
90                          (sizeof(server_ip) /
91                           sizeof(server_ip[0]));
92                  } else if (ip->saddr == server_ip[0] ||
93                         ip->saddr == server_ip[1] ||
94                         ip->saddr == server_ip[2] ||
95                         ip->saddr == server_ip[3]) {
96                      bpf_printk("Response from server%d",
97                              (ip->saddr >> 24) % 10);
98                      __builtin_memcpy(eth->h_dest, client_mac,
99                               sizeof(client_mac));
100                     ip->daddr = client_ip;
101                 }
102                 __builtin_memcpy(eth->h_source, load_balancer_mac,
103                          sizeof(load_balancer_mac));
104                 ip->saddr = load_balancer_ip;
105                 ip->check = iph_csum(ip);
106                 return XDP_TX;
107             }
108         }
109         return XDP_PASS;
110     }
111
112     char _license[] SEC("license") = "GPL";
```

Dopo aver creato un container di Alpine Linux da utilizzare come load balancer, si potrà caricare il programma XDP (è importante utilizzare la modalità generica, tramite `-m skb`, altrimenti tutte le interfacce di rete dei container dovrebbero avere un programma XDP connesso):

```
sudo podman run --rm -it --privileged -v ./:/xdp:Z \
```

```
    --ip 10.88.0.10 \
    --mac-address 02:42:00:00:00:10 \
    --name loadbalancer \
    alpine
apk add xdp-tools
mount -t bpf bpf /sys/fs/bpf/
xdp-loader load eth0 /xdp/xdp_redirect.o -m skb
```

È possibile vedere l'output di `bpf_printk` con:

```
mount -t debugfs none /sys/kernel/debug
cat /sys/kernel/debug/tracing/trace_pipe
```

Gli `xdp-tools` mettono anche a disposizione `xdpdump` che permette di visualizzare il *dump* dei pacchetti in modo da poter vedere anche quelli che vengono ignorati tramite il codice di ritorno `XDP_DROP` o inoltrati con il codice di ritorno `XDP_TX`, cosa che non è possibile fare con `tcpdump`. Per esempio si possono vedere i pacchetti in entrata e in uscita tramite:

```
xdpdump -i eth0 --rx-capture entry,exit -x
```

Per fare i test è possibile utilizzare un container con Alpine Linux in cui installare `curl`:

```
sudo podman run --rm -it \
    --ip 10.88.0.9 \
    --mac-address 02:42:00:00:00:09 \
    --name client \
    alpine
apk add curl
curl 10.88.0.10
```

Si può eliminare l'ambiente di test con:

```
for i in {1..4}; do sudo podman stop "server$i"; done
```

4.4.2 Direct Server Return

Nella precedente implementazione si possono notare due grosse limitazioni:

1. L'indirizzo IP e MAC del client sono *hard coded*: questo è stato fatto soltanto per rendere più semplice l'implementazione di esempio altrimenti sarebbe stato necessario creare una struttura dati in cui memorizzare a quale client inoltrare le risposte provenienti dai server di backend.

2. Tutto il traffico in uscita dai server di backend passa nuovamente per il load balancer.

Il Direct Server Return (DSR), a volte chiamato anche direct routing, è una tecnica che permette ai server di backend di inviare il traffico in uscita direttamente al client evitando così di passare per il load balancer.

Per utilizzare questa tecnica il load balancer e i server di backend devono poter ricevere il traffico indirizzato allo stesso indirizzo IP virtuale. Bisogna però prestare attenzione perché quando si trasmettono pacchetti sulla rete locale, per associare l'indirizzo IP a quello MAC, vengono inviate delle richieste ARP. Se associamo a tutti i server lo stesso indirizzo IP, questi risponderanno alle richieste ARP per quell'indirizzo con il proprio indirizzo MAC e in alcuni casi le richieste potrebbero quindi essere inoltrate direttamente ai server di backend invece che al load balancer. Per evitare ciò si possono usare diverse tecniche:

- Filtrare le richieste ARP sui server di backend tramite `arptables` così solo il load balancer risponderà alle richieste ARP per l'indirizzo IP virtuale.

- Permettere ai server di backend di ricevere pacchetti anche se l'indirizzo IP virtuale non è presente nel sistema, tramite `firewalld` o `iptables`; in questo modo l'indirizzo IP virtuale sarà associato solo al load balancer. Sarà così possibile inoltrare i pacchetti con questo indirizzo anche ai server di backend evitando che questi vengano rifiutati.

- Impostare tramite `sysctl` che i server di backend non annuncino l'indirizzo IP virtuale tramite ARP e non rispondano alle richieste ARP per l'indirizzo virtuale.

Per esempio, per creare i server di backend disattivando, tramite `sysctl`, ARP e aggiungere l'IP virtuale:

```
for i in {1..4}; do
    sudo podman run -d --rm --privileged \
    --sysctl net.ipv4.conf.eth0.arp_ignore=1 \
    --sysctl net.ipv4.conf.eth0.arp_announce=2 \
    --ip "10.88.0.1$i" \
    --mac-address "02:42:00:00:00:1$i" \
    --name "server$i" -h "server$i" \
    nginxdemos/hello:plain-text;
done
for i in {1..4}; do
    sudo podman exec "server$i" ip addr add 10.88.0.10 dev eth0;
done
```

Ora si può modificare il programma XDP per cambiare soltanto gli indirizzi MAC nel frame Ethernet senza modificare il pacchetto IP. I pacchetti verranno inoltrati a un server di backend scelto a seconda dell'istante della richiesta.

Codice sorgente 4.5: Implementazione del Direct Server Return

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <bpf/bpf_endian.h>
6  #include <bpf/bpf_helpers.h>
7
8  #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
9
10 unsigned char load_balancer_mac[ETH_ALEN] = {
11     0x02, 0x42, 0x00, 0x00, 0x00, 0x10
12 };
13
14 unsigned char server_mac[][ETH_ALEN] = {
```

```
15     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x11 },
16     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x12 },
17     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x13 },
18     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x14 },
19 };
20
21 __be32 virtual_ip = IP_ADDRESS(10, 88, 0, 10);
22
23 SEC("xdp")
24 int xdp_dsr(struct xdp_md *ctx)
25 {
26     void *data_end = (void *)(long)ctx->data_end;
27     void *pos = (void *)(long)ctx->data;
28     struct ethhdr *eth = pos;
29     if (eth + 1 > data_end) {
30         return -1;
31     }
32     __u16 h_proto = eth->h_proto;
33     pos = eth + 1;
34     if (h_proto == bpf_htons(ETH_P_IP)) {
35         struct iphdr *ip = pos;
36         if (ip + 1 > data_end) {
37             return -1;
38         }
39         if (ip->protocol == IPPROTO_TCP ||
40             ip->protocol == IPPROTO_UDP) {
41             if (ip->daddr == virtual_ip) {
42                 __builtin_memcpy(eth->h_source,
43                          load_balancer_mac,
44                          sizeof(load_balancer_mac));
45                 __builtin_memcpy(
46                     eth->h_dest,
47                     server_mac[(bpf_ktime_get_ns() /
48                            2000000000) %
49                           4],
50                     sizeof(server_mac[0]));
51                 return XDP_TX;
52             }
```

```
53            }
54        }
55        return XDP_PASS;
56    }
57
58    char _license[] SEC("license") = "GPL";
```

4.4.3 Selezione del server tramite hashing

Nella precedente implementazione il load balancer cambiava il server di backend a cui inoltrare le richieste a seconda dell'istante. In questo caso alcune connessioni si potrebbero interrompere se il server di backend viene cambiato mentre è in corso una connessione, poiché il nuovo server di backend non ha un socket aperto per quest'ultima.

Per evitare questo problema il server di backend va selezionato in modo coerente, per esempio selezionandolo a seconda dell'indirizzo IP e della porta sorgente della connessione. Una tecnica utilizzata è fare l'hashing della tupla formata da: indirizzi IP sorgente e destinazione, porte sorgente e destinazione e protocollo.

Una caratteristica importante che deve avere l'algoritmo di hashing scelto è limitare il numero di riassegnamenti in caso venga aggiunto o rimosso un server di backend. In particolare, in caso di rimozione di un server, solo le connessioni a quel server devono venire riassegnate, mentre devono restare invariate le associazioni agli altri. Nel caso di aggiunta di un server si vuole minimizzare il numero di connessioni che vengono riassegnate e si vogliono evitare modifiche alle assegnazioni ai server già presenti. Queste caratteristiche sono presenti nella tecnica del *consistent hashing*; in particolare è possibile dimostrare che quest'ultima è un caso particolare del rendezvous hashing, più generale e semplice, nel caso di una opzione.

Il rendezvous hashing o highest random weight (HRW) hashing è una tecnica di hashing che permette a dei client di arrivare a un consenso distribuito su k opzioni tra le n possibili. In questo caso si vuole selezionare 1 server/opzione tra gli n server di backend disponibili. L'idea è quella di assegnare a ogni server un punteggio per ogni richiesta e assegnare quella richiesta al server con punteggio maggiore. È possibile assegnare un peso che funge da moltiplicatore a ogni server, per esempio nel caso non tutti i server siano

uguali. Per ogni richiesta r si calcola un punteggio $score_i$ per ogni server s_i tramite una funzione di hash: $score_i = hash(r, s_i)$. Il punteggio viene poi moltiplicato per il peso del server e viene selezionato il server con il punteggio pesato più alto. Se un server viene rimosso, l'algoritmo semplicemente selezionerà il server con il secondo punteggio più alto nei casi in cui il server rimosso era quello con il punteggio più alto, mentre la selezione rimarrà invariata negli altri casi. Come algoritmo di hashing è stato scelto MurmurHash3 [22] poiché è più veloce rispetto ad algoritmi di hashing crittografici.

Codice sorgente 4.6: Utilizzo dell'hashing per la selezione del server

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <linux/tcp.h>
6  #include <linux/udp.h>
7  #include <bpf/bpf_endian.h>
8  #include <bpf/bpf_helpers.h>
9  #include "MurmurHash3.h"
10
11 #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
12
13 #define SERVERS_NUMBER 4
14
15 unsigned char server_mac[][ETH_ALEN] = {
16     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x11 },
17     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x12 },
18     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x13 },
19     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x14 },
20 };
21
22 unsigned int server_weight[] = { 1, 1, 1, 1 };
23
24 __be32 virtual_ip = IP_ADDRESS(10, 88, 0, 10);
25
26 static __always_inline unsigned int server_index_ipv4(__be32 saddr,
27                               __be16 sport,
28                               __be32 daddr,
```

```
29                               __be16 dport)
30 {
31     bpf_printk("sport: %d", bpf_htons(sport));
32     bpf_printk("dport: %d", bpf_htons(dport));
33     unsigned int max_score = 0;
34     unsigned int server_index = 0;
35     for (unsigned int i = 0; i < SERVERS_NUMBER; ++i) {
36         unsigned int score;
37         MurmurHash3_x86_32(&i, sizeof(i),
38                    saddr ^ sport ^ daddr ^ dport, &score);
39         score = score >> 8;
40         score *= server_weight[i];
41         if (score > max_score) {
42             max_score = score;
43             server_index = i;
44         }
45     }
46     bpf_printk("server_index: %d", server_index);
47     return server_index;
48 }
49
50 SEC("xdp")
51 int xdp_hash(struct xdp_md *ctx)
52 {
53     void *data_end = (void *)(long)ctx->data_end;
54     void *pos = (void *)(long)ctx->data;
55     struct ethhdr *eth = pos;
56     if (eth + 1 > data_end) {
57         return -1;
58     }
59     __u16 h_proto = eth->h_proto;
60     pos = eth + 1;
61     if (h_proto == bpf_htons(ETH_P_IP)) {
62         struct iphdr *ip = pos;
63         if (ip + 1 > data_end) {
64             return -1;
65         }
66         pos = ip + 1;
```

```
67          if (ip->protocol == IPPROTO_TCP ||
68              ip->protocol == IPPROTO_UDP) {
69              if (ip->daddr == virtual_ip) {
70                  int server_index;
71                  if (ip->protocol == IPPROTO_TCP) {
72                      struct tcphdr *tcp = pos;
73                      if (tcp + 1 > data_end) {
74                          return -1;
75                      }
76                      server_index = server_index_ipv4(
77                          ip->saddr, tcp->source,
78                          ip->daddr, tcp->dest);
79                  } else {
80                      struct udphdr *udp = pos;
81                      if (udp + 1 > data_end) {
82                          return -1;
83                      }
84                      server_index = server_index_ipv4(
85                          ip->saddr, udp->source,
86                          ip->daddr, udp->dest);
87                  }
88                  __builtin_memcpy(eth->h_source,
89                           eth->h_dest,
90                           ETH_ALEN);
91                  __builtin_memcpy(eth->h_dest,
92                           server_mac[server_index],
93                           ETH_ALEN);
94                  return XDP_TX;
95              }
96          }
97      }
98      return XDP_PASS;
99  }
100
101 char _license[] SEC("license") = "GPL";
```

4.4.4 Impostare i pesi

Una modifica interessante al precedente programma può essere quella di leggere i valori dei pesi da una mappa:

Codice sorgente 4.7: Programma XDP che legge i pesi da una mappa

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <linux/tcp.h>
6  #include <linux/udp.h>
7  #include <bpf/bpf_endian.h>
8  #include <bpf/bpf_helpers.h>
9  #include "MurmurHash3.h"
10
11 #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
12
13 #define SERVERS_NUMBER 4
14
15 struct {
16     __uint(type, BPF_MAP_TYPE_ARRAY);
17     __uint(max_entries, 4);
18     __type(key, __u32);
19     __type(value, long);
20     __uint(pinning, LIBBPF_PIN_BY_NAME);
21 } weights SEC(".maps");
22
23 unsigned char server_mac[][ETH_ALEN] = {
24     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x11 },
25     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x12 },
26     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x13 },
27     { 0x02, 0x42, 0x00, 0x00, 0x00, 0x14 },
28 };
29
30 __be32 virtual_ip = IP_ADDRESS(10, 88, 0, 10);
31
32 static __always_inline unsigned int server_index_ipv4(__be32 saddr,
```

```
33                                 __be16 sport,
34                                 __be32 daddr,
35                                 __be16 dport)
36 {
37     bpf_printk("sport: %d", bpf_htons(sport));
38     bpf_printk("dport: %d", bpf_htons(dport));
39     unsigned int max_score = 0;
40     unsigned int server_index = 0;
41     for (unsigned int i = 0; i < SERVERS_NUMBER; ++i) {
42         unsigned int score;
43         MurmurHash3_x86_32(&i, sizeof(i),
44                    saddr ^ sport ^ daddr ^ dport, &score);
45         score = score >> 8;
46         long *weight;
47         __u32 key = i;
48         weight = bpf_map_lookup_elem(&weights, &key);
49         if (weight) {
50             score *= *weight;
51             bpf_printk("Weight server%d: %u", i, score);
52         }
53         if (score > max_score) {
54             max_score = score;
55             server_index = i;
56         }
57     }
58     bpf_printk("server_index: %d", server_index);
59     return server_index;
60 }
61
62 SEC("xdp")
63 int xdp_hash(struct xdp_md *ctx)
64 {
65     void *data_end = (void *)(long)ctx->data_end;
66     void *pos = (void *)(long)ctx->data;
67     struct ethhdr *eth = pos;
68     if (eth + 1 > data_end) {
69         return -1;
70     }
```

```
71      __u16 h_proto = eth->h_proto;
72      pos = eth + 1;
73      if (h_proto == bpf_htons(ETH_P_IP)) {
74          struct iphdr *ip = pos;
75          if (ip + 1 > data_end) {
76              return -1;
77          }
78          pos = ip + 1;
79          if (ip->protocol == IPPROTO_TCP ||
80              ip->protocol == IPPROTO_UDP) {
81              if (ip->daddr == virtual_ip) {
82                  int server_index;
83                  if (ip->protocol == IPPROTO_TCP) {
84                      struct tcphdr *tcp = pos;
85                      if (tcp + 1 > data_end) {
86                          return -1;
87                      }
88                      server_index = server_index_ipv4(
89                          ip->saddr, tcp->source,
90                          ip->daddr, tcp->dest);
91                  } else {
92                      struct udphdr *udp = pos;
93                      if (udp + 1 > data_end) {
94                          return -1;
95                      }
96                      server_index = server_index_ipv4(
97                          ip->saddr, udp->source,
98                          ip->daddr, udp->dest);
99                  }
100                 __builtin_memcpy(eth->h_source, eth->h_dest,
101                          ETH_ALEN);
102                 __builtin_memcpy(eth->h_dest,
103                          server_mac[server_index],
104                          sizeof(server_mac[0]));
105                 return XDP_TX;
106             }
107         }
108     }
```

```
109     return XDP_PASS;
110 }
111
112 char _license[] SEC("license") = "GPL";
```

In questo modo si possono impostare i pesi dallo user space:

Codice sorgente 4.8: Programma per impostare i pesi da user space

```
 1 #include <bpf/bpf.h>
 2 #include <bpf/bpf.h>
 3 #include <bpf/libbpf.h>
 4 #include <stdio.h>
 5
 6 int main(int argc, char *argv[])
 7 {
 8     int weights = bpf_obj_get("/sys/fs/bpf/test/weights");
 9     if (weights < 0) {
10         printf("Error bpf_obj_get\n");
11         return weights;
12     }
13     for (__u32 server_index = 0; server_index < 4; ++server_index) {
14         printf("Weight server%d: \n", server_index + 1);
15         long weight;
16         scanf("%ld", &weight);
17         if (bpf_map_update_elem(weights, &server_index, &weight,
18                     BPF_EXIST) != 0) {
19             printf("Error bpf_map_update_elem\n");
20             return -1;
21         }
22     }
23     return 0;
24 }
```

È possibile compilare ed eseguire questo programma all'interno del container del load balancer:

```
apk add libbpf-dev linux-headers musl-dev
gcc /xdp/xdp_hash_weight_set.c -o /xdp/xdp_hash_weight_set -lbpf
/xdp/xdp_hash_weight_set
```

Capitolo 5

Testing

5.1 Ambiente di test

Per effettuare i primi test sono state create delle macchine virtuali sul cloud del CNAF tramite OpenStack [23].

È stata creata una macchina virtuale con Rocky Linux 8 da usare come load balancer in cui è stato caricato il programma XDP.

Sono state create 4 macchine virtuali con Rocky Linux 8 dove è stato installato nginx. Tramite `arptables` sono state bloccate le risposte ARP all'IP del load balancer che poi è stato aggiunto a tutti i server di backend (vd. sezione 4.4.2).

```
for i in $(seq 1 4); do
    ssh xdplb-be-$i sudo dnf install nginx -y;
    ssh xdplb-be-$i sudo systemctl enable --now nginx;
    ssh xdplb-be-$i sudo dnf install arptables -y;
    ssh xdplb-be-$i sudo arptables \
        -A INPUT \
        -d <load_balancer_ip> \
        -j DROP;
    realIP=$(getent hosts xdplb-be-$i | awk '{ print $1 }');
    ssh xdplb-be-$i sudo arptables \
        -A OUTPUT -s <load_balancer_ip> \
```

```
        -j mangle \
        --mangle-ip-s $realIP;
    ssh xdplb-be-$i sudo ip addr add <load_balancer_ip> dev eth0;
done
```

È stata poi creata una macchina da utilizzare come client.

Sono state effettuate alcune modifiche al load balancer per evitare che venga bilanciato anche il traffico dei protocolli SSH, DNS e DHCP e per far funzionare le connessioni dal load balancer verso altri server evitando che venga bilanciato il traffico nel caso in cui sia presente il corrispondente socket TCP sul load balancer. È stato sostituito anche l'algoritmo di hash MurmurHash3 con xxHash [24].

Codice sorgente 5.1: Programma XDP usato nei test

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <linux/ipv6.h>
6  #include <linux/tcp.h>
7  #include <linux/udp.h>
8  #include <bpf/bpf_endian.h>
9  #include <bpf/bpf_helpers.h>
10 #define XXH_INLINE_ALL
11 #include "xxhash.h"
12
13 #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
14
15 #define SERVERS_NUMBER 4
16
17 unsigned char server_mac[][ETH_ALEN] = {
18     { 0xfa, 0x16, 0x3e, 0x76, 0x51, 0xf9 },
19     { 0xfa, 0x16, 0x3e, 0x9c, 0x38, 0x17 },
20     { 0xfa, 0x16, 0x3e, 0x01, 0x7d, 0xd9 },
21     { 0xfa, 0x16, 0x3e, 0xc4, 0x77, 0xcb },
22 };
23
```

```
24 unsigned int server_weight[] = { 1, 1, 1, 1 };
25
26 __be32 virtual_ip = IP_ADDRESS(192, 168, 23, 250);
27
28 static __always_inline unsigned int server_index_ipv4(__be32 saddr,
29                                __be16 sport,
30                                __be32 daddr,
31                                __be16 dport)
32 {
33     XXH64_hash_t max_score = 0;
34     unsigned int server_index = 0;
35     for (unsigned int i = 0; i < SERVERS_NUMBER; ++i) {
36         XXH64_hash_t score = XXH3_64bits_withSeed(
37             &i, sizeof(i), saddr ^ sport ^ daddr ^ dport);
38         score = score >> 8;
39         score *= server_weight[i];
40         if (score > max_score) {
41             max_score = score;
42             server_index = i;
43         }
44     }
45     return server_index;
46 }
47
48 SEC("xdp")
49 int xdp_hash(struct xdp_md *ctx)
50 {
51     void *data_end = (void *)(long)ctx->data_end;
52     void *pos = (void *)(long)ctx->data;
53     struct ethhdr *eth = pos;
54     if (eth + 1 > data_end) {
55         return -1;
56     }
57     __u16 h_proto = eth->h_proto;
58     pos = eth + 1;
59     if (h_proto == bpf_htons(ETH_P_IP)) {
60         struct iphdr *ip = pos;
61         if (ip + 1 > data_end) {
```

```
62            return -1;
63        }
64        pos = ip + 1;
65        if (ip->protocol == IPPROTO_TCP ||
66            ip->protocol == IPPROTO_UDP) {
67            if (ip->daddr == virtual_ip) {
68                int server_index = 0;
69                if (ip->protocol == IPPROTO_TCP) {
70                    struct tcphdr *tcp = pos;
71                    if (tcp + 1 > data_end) {
72                        return -1;
73                    }
74                    if (bpf_htons(tcp->dest) == 22 ||
75                        bpf_htons(tcp->source) == 53) {
76                        return XDP_PASS;
77                    }
78                    struct bpf_sock_tuple tuple;
79                    tuple.ipv4.saddr = ip->saddr;
80                    tuple.ipv4.daddr = ip->daddr;
81                    tuple.ipv4.sport = tcp->source;
82                    tuple.ipv4.dport = tcp->dest;
83                    struct bpf_sock *socket =
84                        bpf_sk_lookup_tcp(
85                            ctx, &tuple,
86                            sizeof(tuple.ipv4), -1,
87                            0);
88                    if (socket) {
89                        bpf_sk_release(socket);
90                        return XDP_PASS;
91                    }
92                    server_index = server_index_ipv4(
93                        ip->saddr, tcp->source,
94                        ip->daddr, tcp->dest);
95                } else {
96                    struct udphdr *udp = pos;
97                    if (udp + 1 > data_end) {
98                        return -1;
99                    }
```

```
100                     if (bpf_htons(udp->dest) == 22 ||
101                         bpf_htons(udp->source) == 53 ||
102                         (bpf_htons(udp->source) == 67 &&
103                          bpf_htons(udp->dest) == 68)) {
104                         return XDP_PASS;
105                     }
106                     server_index = server_index_ipv4(
107                         ip->saddr, udp->source,
108                         ip->daddr, udp->dest);
109                 }
110                 __builtin_memcpy(eth->h_source, eth->h_dest,
111                          ETH_ALEN);
112                 __builtin_memcpy(eth->h_dest,
113                          server_mac[server_index],
114                          ETH_ALEN);
115                 return XDP_TX;
116             }
117         }
118     }
119     return XDP_PASS;
120 }
121
122 char _license[] SEC("license") = "GPL";
```

5.2 Risultati

Come primo test sono state effettuate 100000 richieste al secondo per 10 secondi per verificare l'efficienza degli algoritmi di hashing. Visto l'elevato numero di richieste per prima cosa è necessario aumentare il numero di *file descriptor* che un processo può avere.

```
ulimit -n 100000
```

Per generare traffico HTTP è stato utilizzato Vegeta [25].

```
echo "GET http://xdplb-lb/" | \
    ./vegeta attack -duration=10s -rate=100000 | \
    tee results.bin | ./vegeta report
```

Latenza	Senza LB	LB (MurmurHash3)	LB (xxHash)
min	174 µs	205 µs	183 µs
mean	365 µs	410 µs	366 µs
50	285 µs	321 µs	301 µs
90	469 µs	509 µs	424 µs
95	783 µs	832 µs	609 µs
99	1864 µs	2040 µs	1586 µs

Tabella 5.1: Confronto della latenza senza load balancer, con load balancer che utilizza come funzione di hashing MurmurHash3 e con load balancer che utilizza invece xxHash.

Possiamo dedurre che:

- L'aggiunta di un load balancer non influisce significativamente sulla latenza delle richieste.
- La variabilità delle misure è molto ampia ed è dovuta verosimilmente all'infrastruttura di virtualizzazione utilizzata.

Per testare il caso di richieste con un *body* pesante può essere generato un file da 100 MB con:

```
dd if=/dev/urandom of=sample.txt bs=100M count=1 iflag=fullblock
```

Nel blocco `http` del file `/etc/nginx/nginx.conf` nei server di backend è necessario aggiungere il parametro `client_max_body_size 0;` per far accettare da nginx body di qualsiasi dimensione.

Per inviare questo body con Vegeta:

```
echo "GET http://xdplb-lb/" | \
    ./vegeta attack -duration=10s -rate=10 -body=./sample.txt | \
    tee results.bin | \
    ./vegeta report
```

	Senza LB	LB con MurmurHash3	LB con xxHash
Latenza media	111.712 ms	100.203 ms	108.317 ms
Velocità media	7.83 Gpbs	7.82 Gbps	7.82 Gbps
Velocità massima	8.11 Gbps	8.08 Gbps	8.13 Gpbs

Tabella 5.2: La riga latenza media riporta i dati di Vegeta invece le velocità sono ottenute con nload.

I risultati sulla velocità sono stati ottenuti usando `nload`.

Da questi risultati è possibile notare come il load balancer non faccia da collo di bottiglia e sia molto efficiente saturando la banda disponibile.

Capitolo 6

Verso un load balancing distribuito

Ispirandosi a Katran e Unimog il passo successivo naturale rispetto a quanto mostrato nel capitolo precedente è di distribuire il load balancer tra tutti i server di backend, evitando di ospitarlo su una macchina dedicata. Secondo questo approccio il router distribuisce i pacchetti direttamente ai server, che si fanno carico di inoltrarli al server corretto. In questo modo si evita che la macchina con il load balancer faccia da collo di bottiglia anche nel caso del traffico in ingresso.

Nelle sezioni seguenti è illustrato come è stato avviato lo sviluppo del modello succitato.

6.1 Ambiente virtuale

Per creare un ambiente virtuale in cui simulare un router a cui sono collegati dei server è stato usato GNS3 [26].

Come router è stato usato un Cisco Catalyst 8000V. La porta GigabitEthernet4 è stata collegata al NAT in modo da poter collegarsi alla rete Internet. Le due porte GigabitEthernet1 e GigabitEthernet2 sono collegate a due switch, mentre la GigabitEthernet3 è collegata a un webterm. Il router è stato configurato nel seguente modo.

```
enable
configure terminal
interface gigabitEthernet 4
```

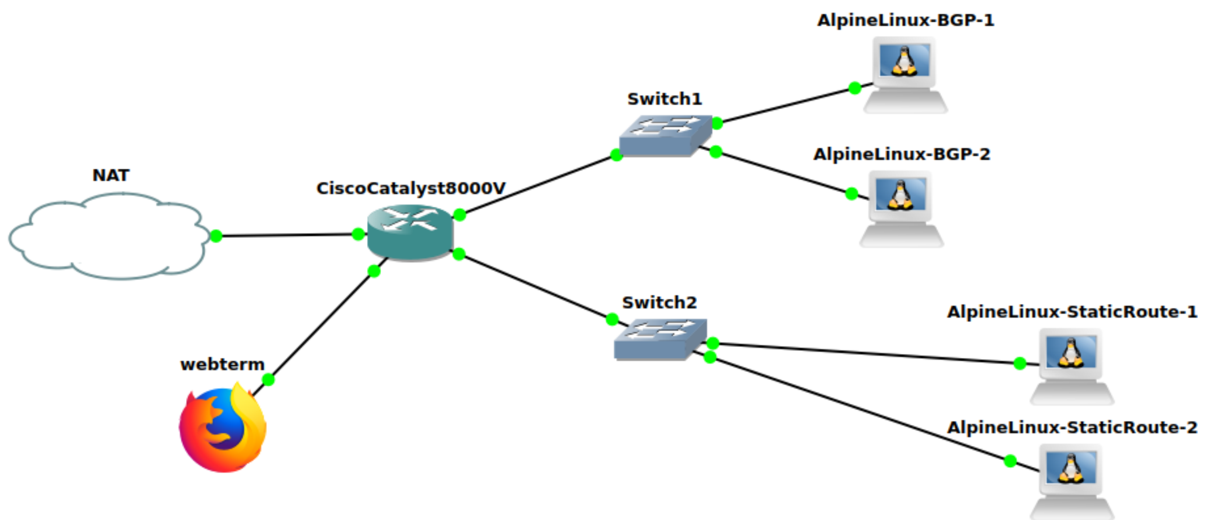


Figura 6.1: Topologia di rete creata usando GNS3

```
 ip nat outside
interface gigabitEthernet 1
 ip address 10.10.10.254 255.255.255.0
 no shutdown
 ip nat inside
ip nat inside source list 1 interface gigabitEthernet 4 overload
access-list 1 permit 10.0.0.0 0.255.255.255
interface gigabitEthernet 2
 ip address 10.20.20.254 255.255.255.0
 no shutdown
 ip nat inside
interface gigabitEthernet 3
 ip address 10.30.30.254 255.255.255.0
 no shutdown
 ip nat inside
ip dhcp pool webterms
 network 10.30.30.0 255.255.255.0
 default-router 10.30.30.254
```

```
 dns-server 1.1.1.1
```

È stato configurato BGP nel router in modo da permettere ai server di annunciare un indirizzo IP virtuale:

```
interface Loopback0
 ip address 192.168.124.1 255.255.255.255
router bgp 65000
 neighbor 10.10.10.1 remote-as 10
 neighbor 10.10.10.2 remote-as 10
 network 192.168.124.1 mask 255.255.255.255
 maximum-paths 2
```

Sono stati collegati al primo switch due container con Alpine Linux da usare come server e configurata la rete, per esempio nel caso del primo server nel seguente modo:

```
auto eth0
iface eth0 inet static
    address 10.10.10.1
    netmask 255.255.255.0
    gateway 10.10.10.254
    up echo nameserver 1.1.1.1 > /etc/resolv.conf
```

Dopo aver installato ExaBGP, si deve creare un file di configurazione `config.ini`:

```
neighbor 10.10.10.254 {
    router-id 10.10.10.1;
    local-address 10.10.10.1;
    local-as 10;
    peer-as 65000;
}
```

Dopo aver avviato ExaBGP, è possibile annunciare tramite BGP l'indirizzo IP virtuale e aggiungerlo alla scheda di rete dopo aver impedito che venga annunciato tramite ARP:

```
exabgp ./config.ini &
exabgpcli announce route 192.168.125.1/32 next-hop 10.10.10.1
sysctl net.ipv4.conf.eth0.arp_ignore=1
sysctl net.ipv4.conf.eth0.arp_announce=2
ip addr add 192.168.125.1 dev eth0
```

Dopo aver installato ngnix, per poter visualizzare quale server risponde alla richiesta, è necessario modificare il file di configurazione `/etc/nginx/http.d/default.conf`:

```
location / {
    default_type text/plain;
    expires -1;
    return 200 '$server_addr:$server_port\nServer name: $hostname\n';
}
```

Un altro modo per assegnare lo stesso IP a più server è utilizzare le rotte statiche. Il principale vantaggio di questa tecnica rispetto a BGP è che non bisogna installare nulla sui server, né configurare BGP sul router. Al contrario di BGP però, se uno dei server ha dei problemi di connessione o viene spento, il router non ha modo di accorgersene e continuerà a cercare di inviargli i pacchetti. Con BGP inoltre è possibile fare il *withdraw* della rotta ed è possibile configurare ExaBGP in modo che lo faccia in automatico nel caso alcuni servizi sul server non funzionino.

Al secondo switch sono stati collegati altri due container con Alpine Linux configurati similmente ai precedenti però in questo caso, invece di utilizzare BGP, l'IP virtuale è stato configurato utilizzando delle rotte statiche:

```
ip route 192.168.126.1 255.255.255.255 10.20.20.1
ip route 192.168.126.1 255.255.255.255 10.20.20.2
```

Alla terza porta è stato collegato un webterm con la seguente configurazione che sfrutta il pool DHCP configurato nel router:

```
auto eth0
iface eth0 inet dhcp
    hostname webterm
```

6.2 Load balancer XDP distribuito

È ora possibile compilare e caricare il seguente programma XDP (dopo averlo modificato inserendo gli indirizzi MAC e IP corretti):

Codice sorgente 6.1: Load balancer che utilizza XDP installabile direttamente sui server

```
1  #include <linux/bpf.h>
2  #include <linux/if_ether.h>
3  #include <linux/in.h>
4  #include <linux/ip.h>
5  #include <linux/tcp.h>
6  #include <linux/udp.h>
7  #include <bpf/bpf_endian.h>
8  #include <bpf/bpf_helpers.h>
9  #include "MurmurHash3.h"
10
11 #define IP_ADDRESS(a, b, c, d) (__be32)(a+(b<<8)+(c<<16)+(d<<24))
12
13 #define SERVERS_NUMBER 2
14
15 unsigned char server_mac[][ETH_ALEN] = {
16     { 0xfa, 0xb4, 0x86, 0x7b, 0xcc, 0xf8 },
17     { 0x8a, 0xde, 0xef, 0xf5, 0x26, 0x1a },
18 };
19
20 unsigned int server_weight[] = { 1, 1 };
21
22 __be32 virtual_ip = IP_ADDRESS(192, 168, 125, 1);
23
24 static __always_inline unsigned int server_index_ipv4(__be32 saddr,
25                               __be16 sport,
26                               __be32 daddr,
27                               __be16 dport)
28 {
29     unsigned int max_score = 0;
30     unsigned int server_index = 0;
31     for (unsigned int i = 0; i < SERVERS_NUMBER; ++i) {
32         unsigned int score;
```

```
33          MurmurHash3_x86_32(&i, sizeof(i),
34                     saddr ^ sport ^ daddr ^ dport, &score);
35          score = score >> 8;
36          score *= server_weight[i];
37          if (score > max_score) {
38              max_score = score;
39              server_index = i;
40          }
41      }
42      return server_index;
43  }
44
45  SEC("xdp")
46  int xdp_lb(struct xdp_md *ctx)
47  {
48      void *data_end = (void *)(long)ctx->data_end;
49      void *pos = (void *)(long)ctx->data;
50      struct ethhdr *eth = pos;
51      if (eth + 1 > data_end) {
52          return -1;
53      }
54      __u16 h_proto = eth->h_proto;
55      pos = eth + 1;
56      if (h_proto == bpf_htons(ETH_P_IP)) {
57          struct iphdr *ip = pos;
58          if (ip + 1 > data_end) {
59              return -1;
60          }
61          pos = ip + 1;
62          if (ip->protocol == IPPROTO_TCP ||
63              ip->protocol == IPPROTO_UDP) {
64              if (ip->daddr == virtual_ip) {
65                  int server_index;
66                  if (ip->protocol == IPPROTO_TCP) {
67                      struct tcphdr *tcp = pos;
68                      if (tcp + 1 > data_end) {
69                          return -1;
70                      }
```

```
71                     server_index = server_index_ipv4(
72                         ip->saddr, tcp->source,
73                         ip->daddr, tcp->dest);
74                 } else {
75                     struct udphdr *udp = pos;
76                     if (udp + 1 > data_end) {
77                         return -1;
78                     }
79                     server_index = server_index_ipv4(
80                         ip->saddr, udp->source,
81                         ip->daddr, udp->dest);
82                 }
83                 if (__builtin_memcmp(eth->h_dest,
84                              server_mac[server_index],
85                              sizeof(eth->h_dest)) ==
86                     0) {
87                     return XDP_PASS;
88                 } else {
89                     __builtin_memcpy(eth->h_source,
90                              eth->h_dest,
91                              sizeof(eth->h_dest));
92                     __builtin_memcpy(
93                         eth->h_dest,
94                         server_mac[server_index],
95                         sizeof(server_mac[0]));
96                     return XDP_TX;
97                 }
98             }
99         }
100     }
101     return XDP_PASS;
102 }
103
104 char _license[] SEC("license") = "GPL";
```

Utilizzando `curl` dal webterm si può verificare che il traffico verso `192.168.125.1` viene bilanciato tra i due server collegati al primo switch che annunciano l'IP virtuale tramite BGP. Nello stesso modo è possibile controllare che il traffico verso `192.168.126.1`

viene distribuito tra i server connessi al secondo switch con la configurazione dell'IP virtuale tramite rotte statiche.

Questa implementazione può essere ampliata leggendo i valori dei pesi dalle mappe e usando il daisy chaining per salvare la cronologia delle modifiche in modo da mantenere valide le connessioni vecchie in caso di cambiamenti. Un'altra modifica potrebbe essere quella di accettare i pacchetti se sul server è presente un socket per quella connessione, per esempio usando la funzione `bpf_sk_lookup_tcp`. In questa implementazione però bisogna considerare l'*overhead* dovuto all'inoltro dei pacchetti, potenzialmente anche più volte nel caso di connessioni *long-lived*.

Conclusioni e sviluppi futuri

In questa tesi è stato mostrato come un approccio al load balancing basato su eXpress Data Path (un framework per eBPF che permette di processare i pacchetti di rete in modo efficiente) possa essere promettente, sia in termini di flessibilità che di prestazioni.

Dai test preliminari effettuati su un'infrastruttura virtualizzata messa a disposizione dal CNAF (il centro nazionale delle tecnologie informatiche dell'INFN), si è osservato che l'introduzione del load balancer implementato con XDP, integrato con l'utilizzo del Direct Server Return, non influenza in modo misurabile né la latenza né la velocità dei trasferimenti di rete, riuscendo a saturare i collegamenti disponibili a 10 Gbps. Le prestazioni possono essere incrementare ulteriormente utilizzando le modalità Native e Offloaded, nel caso la scheda di rete la supporti.

L'implementazione proposta non impatta sui servizi già presenti nei server di backend e non richiede modifiche all'architettura di rete.

L'approccio al load balancing investigato in questa tesi si presta a promettenti sviluppi futuri, a partire dalla distribuzione del servizio di load balancing implementato con XDP sui server di backend, evitando il collo di bottiglia anche in ingresso.

Bibliografia

[1] ISO, "Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model," International Organization for Standardization, Standard ISO/IEC 7498-1:1994, Jun. 1996. [Online]. Available: https://www.iso.org/standard/20269.html

[2] "nginx." [Online]. Available: https://nginx.org

[3] Y. Rekhter, S. Hares, and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4271

[4] T. Julienne, "GLB: GitHub's open source load," 2018. [Online]. Available: https://github.blog/2018-08-08-glb-director-open-source-load-balancer/

[5] T. Herbert, L. Yong, and O. Zia, "Generic UDP Encapsulation," Internet Engineering Task Force, Internet-Draft draft-ietf-intarea-gue-09, Oct. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-intarea-gue/09/

[6] "DPDK." [Online]. Available: https://www.dpdk.org

[7] R. D. Nikita Shirokov, "Open-sourcing Katran, a scalable network load balancer," 2018. [Online]. Available: https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[8] "ExaBGP." [Online]. Available: https://github.com/Exa-Networks/exabgp

[9] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein,

"Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, 2016, pp. 523–535. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[10] "Program Types: XDP." [Online]. Available: https://docs.cilium.io/en/stable/bpf/progtypes/#xdp

[11] D. Wragg, "Unimog - Cloudflare's edge load balancer," 2020. [Online]. Available: https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/

[12] "Program Types: tc (traffic control)." [Online]. Available: https://docs.cilium.io/en/stable/bpf/progtypes/#tc-traffic-control

[13] "HashiCorp Consul." [Online]. Available: https://www.hashicorp.com/products/consul

[14] "Prometheus." [Online]. Available: https://prometheus.io

[15] "What is eBPF?" [Online]. Available: https://ebpf.io/what-is-ebpf/

[16] "eBPF Instruction Set Specification, v1.0." [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/instruction-set.html

[17] "Clang." [Online]. Available: https://clang.llvm.org

[18] "eBPF Updates #4: Did You Know? Program size limit." [Online]. Available: https://ebpf.io/blog/ebpf-updates-2021-02/#did-you-know-program-size-limit

[19] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[20] "libbpf." [Online]. Available: https://github.com/libbpf/libbpf/

[21] "Test environment script." [Online]. Available: https://github.com/xdp-project/xdp-tutorial/tree/master/testenv

[22] "MurmurHash3." [Online]. Available: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

[23] "OpenStack." [Online]. Available: https://www.openstack.org

[24] "xxHash." [Online]. Available: https://github.com/Cyan4973/xxHash

[25] "Vegeta." [Online]. Available: https://github.com/tsenart/vegeta

[26] "GNS3." [Online]. Available: https://gns3.com

Ringraziamenti

Per primi vanno sicuramente ringraziati i miei genitori, nonché finanziatori, senza i quali non sarei letteralmente qui, mio fratello e i nonni.

Un grazie speciale ai coinquilini che hanno reso questi anni a Bologna fantastici e indimenticabili (in ordine circa cronologico): "mamma" Silvia, "queen" Greta, Francesco, Jacopo, Chiara, Marco, che è stato un compagno di stanza fantastico, Sara e Valentina.

Ringrazio i miei relatori di tesi, Renzo Davoli e Francesco Giacomini, che mi hanno seguito nello sviluppo di questa tesi. Durante la stesura ho passato due mesi al CNAF, dove ho conosciuto persone gentilissime, in particolare: Antonio Falabella, che mi ha seguito durante il mio periodo da Summer Student e ha revisionato la tesi, Lorenzo Chiarelli, per l'aiuto nella configurazione delle reti e nella revisione delle sezioni sul networking, Diego Michelotto, per avermi messo a disposizioni le macchine virtuali usate nei test, Carmelo "budda" Pellegrino, che mi ha aiutato in troppe cose per essere elencate tutte qui, e tutti gli altri ricercatori e tecnologi che ho avuto il piacere di conoscere.