POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Monitoraggio delle prestazioni di reti a pacchetto con IOVisor

Relatore

Prof. Fulvio Risso

Candidato

Fabio Salvini

ANNO ACCADEMICO 2017/2018

Indice

1 Introduzione

Negli ultimi anni ogni Internet Service Provider (ISP) ha dovuto fronteggiare nuove sfide per adattarsi ai cambiamenti legati al mondo delle comunicazioni delle reti a pacchetto, focalizzando la propria attenzione in due specifiche direzioni:

- la tipologia del traffico trasportato, che, ad oggi, è principalmente legato ai contenuti video.
- l'aumento e la differenziazione della propria offerta in termini di servizi forniti ai clienti.

Per entrambe le tematiche è importante che l'utente finale goda della migliore esperienza possibile ed è quindi necessario che il Service Provider abbia a disposizione dei metodi, e degli strumenti, per poter monitorare e misurare le prestazioni della propria rete con un adeguato tasso di precisione. Tali tools di monitoraggio non servirebbero solo a valutare parametri quali packet loss, delay e jitter ma anche per migliorare la gestione dell'intera rete, potendo, ad esempio, individuare eventuali problematiche in una sua sotto porzione.

In quest'ottica TIM, a causa di tecniche non adeguate a valutare la percentuale di pacchetti persi all'interno della propria rete, ha progettato una nuova metodologia per il monitoraggio passivo del traffico[1], che prende il nome di Packet Network Performance Monitoring (PNPM). PNPM non richiede alcune estensione o modifica dei protocolli esistenti, evitando quindi problemi di interoperabilità, ed è potenzialmente applicabile ad ogni tipo di pacchetto, che sia IP o MPLS, unicast o multicast. TIM ha depositato, a partire dal 2008, una serie di richieste di brevetto relative a PNPM e ne è stata proposta la standardizzazione con un draft presentato in IETF [1]. Tale documento ha destato un certo interesse, che ha portato successivamente alla stesura di altri draft, in collaborazione con Huawei e Cisco [2] [3]. L'implementazione di PNPM presenta però alcune problematiche quali il processamento dei pacchetti a velocità di linea e l'acquisizione dei timestamp.

[1] Le misure passive, al contrario di quelle attive, non richiedono la generazione di traffico artificiale per la valutazione delle performance della rete.

1 – Introduzione

Nel corso del 2014 il kernel Linux, versione 3.15, ha goduto di un'importante aggiornamento, estendendo le capacità, dell'ormai ben consolidato, Berkeley Packet Filter [4] (BPF). Tale componente viene sfruttato dalle librerie Libpcap [5], Winpcap [6] e tools quali Tcpdump [5] e Wireshark [7], strumenti che fanno uso di una virtual machine, posta lato kernel, per effettuare delle operazioni di filtraggio del traffico. Con le nuove capacità, che vengono continuamente aggiunte con l'uscita di nuovi aggiornamenti, è ora possibile svolgere nuovi tipi di operazioni legate ai mondi delle network functions, tracing e security, in ambiente isolato e che non mina la sicurezza del sistema.

Con l'introduzione di questo "extended-BPF" [8] (eBPF) sono nati diversi progetti per accelerarne lo sviluppo e mostrarne le ampie capacità. Fra questi non si può non menzionare IOVisor [9], progetto open source all'interno della comunità Linux che segue la scia delle innovazioni legate al mondo eBPF, al fine di facilitare la scrittura, la compilazione e l'iniezione a run-time dei programmi eBPF nella virtual machine.

Sfruttando tali strumenti è stato possibile creare un prototipo di un'implementazione di PNPM basata su eBPF al fine di affrontare e superare le problematiche, che verranno meglio esposte nel prossimo capitolo.

2 Problema

2.1 Implementazione PNPM

L'obiettivo primario per cui fu ideato PNPM è il riconoscimento dei flussi da monitorare e il conteggio dei pacchetti, al fine di rilevare eventuali perdite. Queste operazioni possono essere svolte con facilità dai router, tramite la scrittura di alcuni script. Tale soluzione appare valida e a costo zero. Volendo però estendere PNPM alla misura dei valori temporali insorgono dei problemi, in quanto non tutti i device fornisco i timestamp dei pacchetti.

A fronte di questi limiti si è deciso quindi di realizzare delle applicazioni ad hoc per il monitoraggio e migrare l'elaborazione sui server. Questo approccio fa però insorgere alcuni nuovi dubbi:

- un "normale" programma in user space avrebbe diverse difficoltà ad elaborare il traffico a velocità di linea e le prestazioni potrebbero essere basse.
- nell'ambito dei data center non è possibile realizzare un proprio componente dedicato al monitoraggio lato kernel in quanto, tipicamente, l'hypervisor[1] è chiuso ad ogni modifica e/o estensione di terze parti.

L'uso di eBPF e della relativa virtual machine lato kernel sembra quindi idoneo nel nostro contesto, affetto dai vincoli che sono stati appena evidenziati. L'implementazione di PNPM prevede quindi la scrittura di programmi eBPF atti a intercettare i pacchetti ricevuti, riconoscere i flussi e raccogliere statistiche. Tutto questo senza la necessità di creare delle versioni custom del kernel, poiché la virtual machine è integrata nell'ambiente Linux.

[1] L'hypervisor fornisce un'astrazione delle risorse hardware ai sistemi operativi e alle applicazioni, garantendone l'isolamento.

2.2 Obiettivi tesi

Il focus primario della tesi è rivolto ovviamente alla realizzazione del programma eBPF per il monitoraggio delle prestazioni di rete ed un componente in user space per la lettura e l'esportazione delle misure raccolte. Si valuteranno quindi i risultati raggiunti effettuando dei test di carico per stimare il numero di pacchetti processabili e il relativo throughput.

Un secondo obiettivo riguarda invece la combinazione della metodologia PNPM con alcune tecniche di campionamento dei pacchetti, al fine di ottimizzare le prestazioni. Questo ha portato alla scrittura di diversi programmi eBPF, le cui performance verranno messe a confronto con la prima versione. Infine si svolgerà un'analisi su quanto ottenuto cercando di comprenderne le cause, limitazioni e possibili soluzioni.

Da qui in avanti verrà usata la dicitura IOVisor-PNPM per indicare sia i programmi di tracing eBPF, iniettati nel kernel, sia il programma in user space per la gestione delle misure raccolte.

3 Background

In questo capitolo si andranno a descrivere i principi base di PNPM, delle tecniche legate al campionamento dei pacchetti, in particolare dell'hash sampling, e le tecnologie usate per la realizzazione dei programmi di monitoraggio.

3.1 PNPM

PNPM descrive un procedimento con il quale è possibile effettuare il monitoraggio delle prestazioni in reti a commutazione di pacchetto. Tale metodologia permette di rilevare eventuali perdite di pacchetti e ricavare misure relative al delay e al jitter.

Aspetti chiave di PNPM:

- Per individuare eventuali packet loss PNPM utilizza un approccio alternativo alla classica numerazione dei singoli pacchetti. Questa tecnica si basa sul conteggio dei pacchetti ricevuti in un dato intervallo temporale e in un successivo confronto dei valori ottenuti. Tale metodologia presenta il vantaggio di non dover leggere/inserire un identificativo univoco, con una conseguente riduzione del tempo di processamento per ogni pacchetto. Di contro, risulta necessaria una sincronizzazione tra gli apparati che effettuano le misure, affinché queste ultime sia relative allo stesso insieme di pacchetti. Tale sincronizzazione può essere effettuata tramite protocolli quali NTP e PTP.

- PNPM divide virtualmente il flusso di pacchetti in blocchi consecutivi, marcandone i pacchetti. I pacchetti appartenenti ad ogni blocco avranno ovviamente la stessa marcatura, mentre tra blocchi consecutivi sarà diversa. Usando tale espediente è possibile individuare il set di pacchetti su cui effettuare le misure e poterle confrontare, in modo coerente, con quelle degli altri apparati. Per

semplicità i blocchi sono marcati in modo alternato, in modo da dover utilizzare solamente due contatori per ogni flusso considerato. PNPM non fornisce indicazioni precise su come effettuare la marcatura dei pacchetti, suggerendo però l'uso del campo DSCP (o TOS, seconda le vecchia dicitura) dell'IP header. Si fa notare inoltre che l'alternanza nella marcatura tra un blocco ed il successivo rappresenta una sorta di segnale di auto-sincronizzazione, garantendo la consistenza nelle misure prese dai differenti apparati. La durata temporale del singolo blocco è detta *Measurement Period* (MP) mentre la durata dell'intero test viene denominata *Marking Cycle Period (MPC).*

- Questa "suddivisione" del traffico evita alcune problematiche legate alla lettura dei contatori, in quanto sarebbe difficile stabilire con precisione l'instante in cui andare a effettuare tale operazione senza che vi sia un errore di sfasamento tra il valore corrente della variabile ed il reale numero di pacchetti ricevuti. Da queste osservazioni si deduce che il contatore andrà letto quando la misura è stabile, ovvero quando il blocco, a cui fa riferimento il contatore, è terminato. La lettura viene comunque fatto dopo un "tempo di guardia" dal termine del blocco, al fine di conteggiare correttamente eventuali pacchetti ritardatari. Il numero di contatori necessari è pari a due, poiché due sono i possibili valori della marcatura, per ogni interfaccia e per ogni flusso.

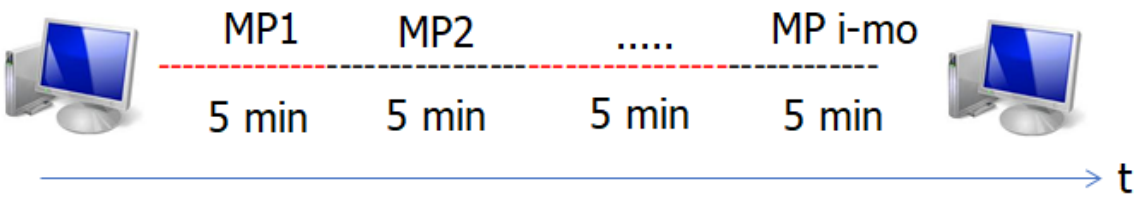


Figura 1 Suddivisione flusso in blocchi colorati.

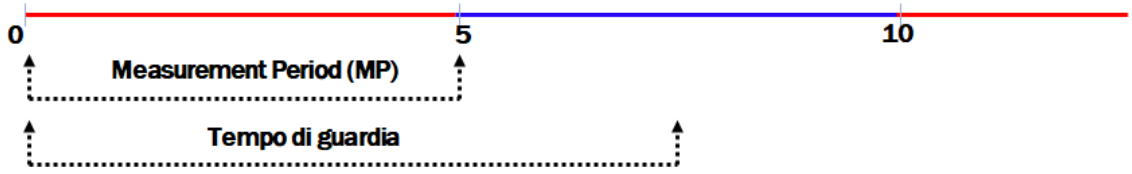


Figura 2 Durata del singolo blocco e tempo di guardia per la lettura del contatore.

- Questo approccio può essere utilizzato anche per ricavare il delay one-way e il jitter one-way. All'arrivo di ogni pacchetto sarebbe infatti possibile memorizzare il timestamp per poi confrontarlo con quello calcolato dagli altri apparati. Tale metodologia funziona però solamente se non vi è stato packet loss o "out-of-order delivery". Infatti il pacchetto N-esimo ricevuto dal primo apparato potrebbe essere diverso da quello ricevuto dagli altri. Al fine di superare tali limitazioni PNPM abbandona le misure puntali sui pacchetti per sostituirle con delle misure medie.

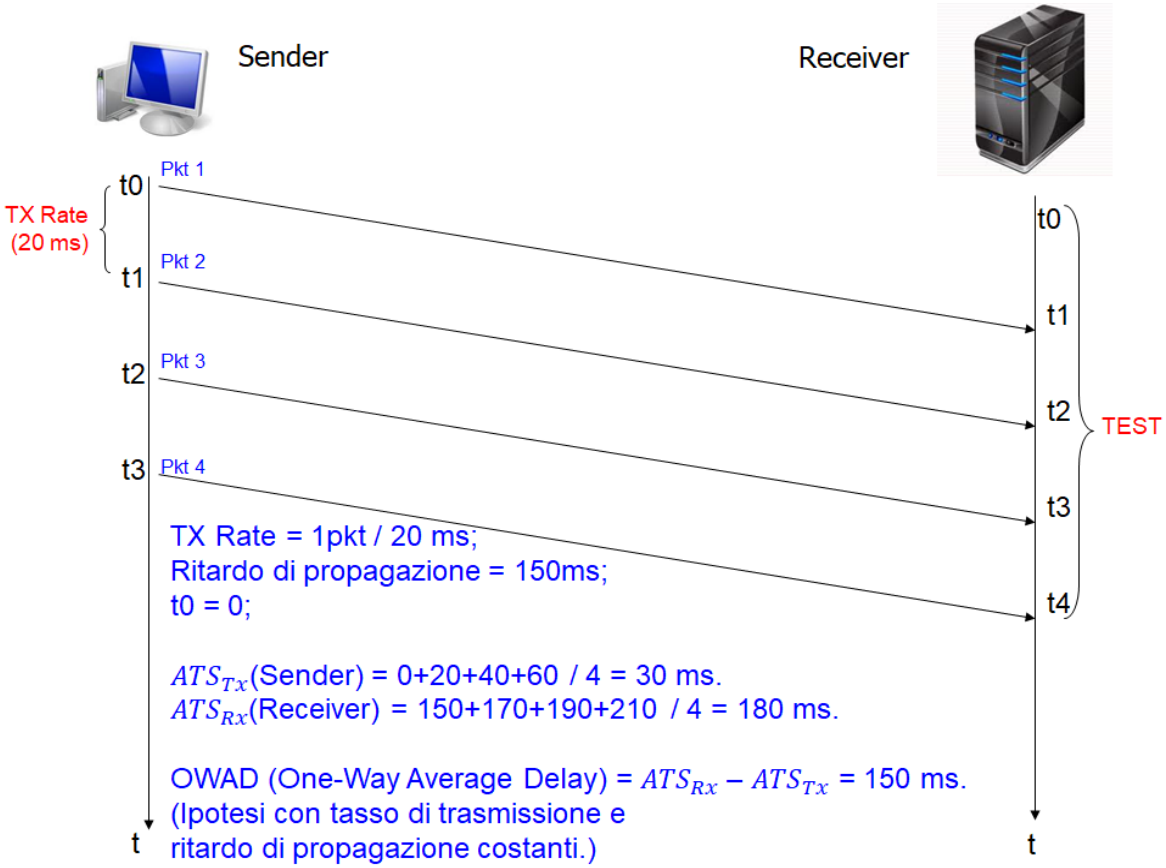


Figura 3 Esempio calcolo misure one-way.

- Viene quindi introdotto il concetto di A*verage TimeStamp* (ATS) per poter ricavare il O*ne-Way Average Delay* (OWAD), da cui a sua volta è possibile ricavare *l'Average Round Trip Time* (ARTT). Per analogia possiamo quindi parlare di O*ne-Way Average Jitter*(OWAJ) e T*wo-Way Average Jitter*(TWAJ).

Average TimeStamp (ATS) = $\sum_{k=1}^{N} TS_i$
One-Way Average Delay (OWAD) = $ATS_{Rx} - ATS_{Tx}$
Average Round Trip Time (ARTTT) = $OWAD_{Tx \to Rx} + OWAD_{Rx \to Tx}$
One-Way Average Jitter (OWAJ) = $\sum_{k=1}^{N} \frac{-1(TS_{i+1} - TS_i)}{N-1} = \frac{TS_N - TS_1}{N-1}$
Two-Way Average Jitter (TWAJ) = $OWAD_{Tx}$ - $OWAD_{Rx}$

Figura 4 Formule per il calcolo delle misure medie.

Per quanto è stato detto si può concludere che

- PNPM è potenzialmente applicabile ad ogni tipo di traffico, sia unicast che multicast. Non richiede l'estensione di protocolli già esistenti ne interagisce con essi. E'quindi *technology and vendor independent*.

- PNPM prevede la possibilità di effettuare sia misure end-2-end, quindi tra i nodi terminali della sessione di monitoraggio, sia misure hop-by hop, quindi tra nodi intermedi, appartenenti al percorso seguito dal flusso dati tra i due nodi terminali.

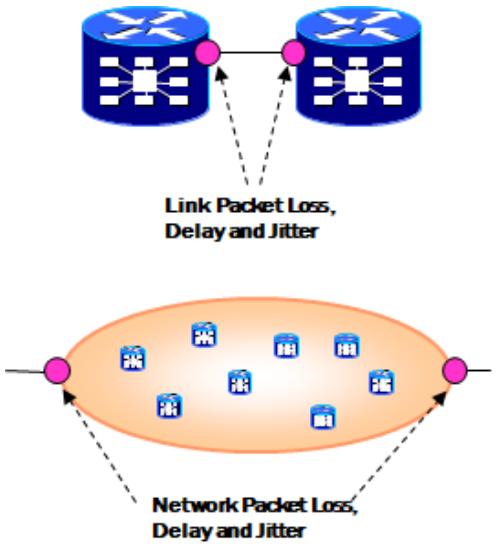


Figura 5 Tipologie misure realizzabili.

3.2 Packet samling e hash-based sampling

Il salvataggio delle misure sul traffico ad alta velocità può essere difficoltoso, specialmente in termini di numero di informazioni raccolte che dovranno essere elaborate successivamente. Il Packet SAMPling [10] (PSAMP) descrive alcune tecniche per poter selezionare un sottoinsieme di pacchetti da quelli ricevuti, per poi produrre successivamente un report. L'hash-based sampling rappresenta una possibile metodologia di campionamento dei pacchetti.

Tale metodo consiste nell'applicazione di una funzione di hash ad ogni pacchetto in arrivo, selezionando alcuni campi da esso per calcolare l'hash. Se il valore ottenuto dalla funzione coincide con un valore prestabilito allora il pacchetto verrà analizzato e usato per il successivo report. Seguendo questo approccio si può dedurre che il sottoinsieme dei pacchetti selezionati da un determinato flusso, in due o più punti di misura, che utilizzino la medesima funzione di hash e i medesimi parametri di configurazione, è uguale per tutti gli apparati considerati, a meno di packet loss. Questa caratteristica, che prende il nome di *consistent selection* [11] permette un confronto diretto fra i report degli apparati di misura, svolto ad esempio da un sistema centrale di raccolta dati.

Vengono qui brevemente analizzate le funzioni di hash e i campi del pacchetto usati da queste ultime.

3.2.1 Funzioni di hash

Come per tutte le funzioni di hash, anche quelle usate per le operazioni di packet sampling devono godere di alcune proprietà per essere considerate idonee, tra cui:

1. Uniformità nella distribuzione dei risultati
2. Poche/nulle collisioni
3. Lunghezza del valore di hash

4. Velocità di esecuzione

Nel contesto del campionamento le proprietà 1 e 4 risultano particolarmente cruciali nella scelta della funzione.

PSAMP indica tre possibili funzioni di hash: IPSX, CRC32 e BOB. Verranno qui brevemente discusse le funzioni IPSX e BOB poiché sono state implementate entrambe nei programmi di tracing. CRC32 non è stata presa in considerazione in quanto i risultati dei test prestazionali [11] non hanno evidenziato un ambito in cui CRC32 predomini rispetto altre funzioni.

- **IPSX**: presi i campi del pacchetto come input vengono fatte alcune operazioni di shift e xor per arrivare a produrre il valore di hash finale su 32 bit, di cui però i primi 16 sono molto simili fra loro. Viene quindi tipicamente detto che la funzione produce un hash su 16bit, determinando un range teorico di valori tra $[0, 2^{16} -1]$.

- **BOB**: questa funzione di hash, realizzata da Bob Jenkins, è stata progettata in modo che ogni bit dell'input influenzi ogni bit del valore finale. BOB utilizza sia i campi del pacchetto sia dei *magic numbers*, valori arbitrari su 32bit, per calcolare il risultato finale. L'hash, su 32bit viene ottenuto applicando una funzione di "mix", nella quale vengono effettuate diverse operazioni aritmetiche su ognuno byte dei campi usati come input e dei magic numbers.

E'stato evidenziato [11] che sebbene IPSX sia la funzione dai costi computazionali più bassi, grazie all'uso delle sole operazioni di shift e xor, questa produca dei risultati scarsamente distribuiti. La funzione di hash BOB appare invece leggermente più lenta di IPSX poiché usa anche altre operazioni aritmetiche. Quest'ultima possiede però una distribuzione dei dati molto più uniforme rispetto a quella di IPSX. Per queste motivazioni PSAMP suggerisce l'uso della funzione di hash BOB, mentre segnala come opzionali le altre funzioni.

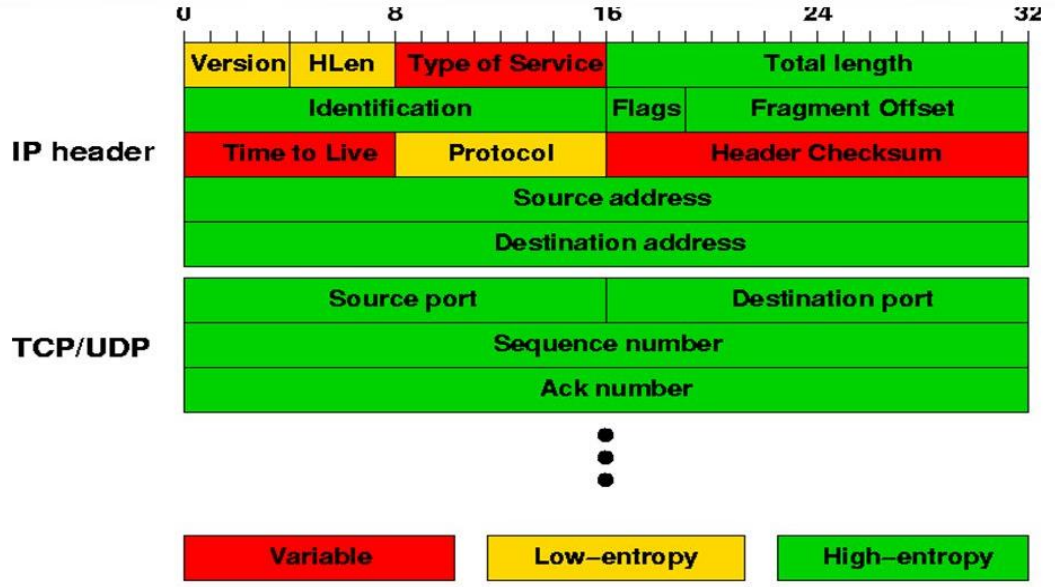


Figura 6 Campi del pacchetto invariabili e ad alta entropia – Rif. [12].

3.2.2 Campi del pacchetto usati dalla funzione di hash

La scelta dei campi dal pacchetto non è affatto casuale, poiché questi devono godere di due proprietà:

- Il valore del campo deve restare invariato durante tutto il percorso del pacchetto. Questo esclude l'uso di campi quali il TTL, che vengono/possono essere modificati dagli apparati.
- I valori assumibili dal campo devono avere un'alta variabilità, al fine di garantire una buona uniformità ai risultati della funzione di hash.

PSAMP suggerisce [10] l'uso dei seguenti campi per le funzioni di hash:

- IP Identification
- Flags
- Fragment offset
- Source IP
- Destination IP

- Numero variabile di byte dal IP payload. Tipicamente vengono scelti il TCP sequence number o la coppia UDP length e UDP checksum, a seconda dei casi.

A questo elenco è stato aggiunto il campo IP Total length, in quanto è stato dimostrato [13] che per la funzione di hash BOB, nel caso peggiore, sono necessari 20B di input per ottenere dei risultati uniformemente distribuiti.

3.2.3 PNPM e hash sampling

TIM ha pensato di combinare le capacità dell'hash sampling e della metodologia PNPM, con l'obiettivo di processare il traffico a velocità di rete e di ridurre il carico elaborativo sugli apparati, in quanto vi sarebbe un alto costo computazionale [14] nell'acquisizione dei timestamp dei pacchetti. Scelto un valore di riferimento questo verrà usato per filtrare i pacchetti in base al loro hash, ottenuto con un'opportuna fuzione. Poiché l'uso di uno specifico valore sarebbe un vincolo estremamente forte e si rischierebbe di selezionare pochissimi pacchetti, si usano solo i bit meno significati del valore di riferimento e dell'hash per effettuare il confronto. Mediamente, fissato il valore di un singolo bit, si scarta quindi circa il 50% dei pacchetti. Il numero dei bit usati per il confronto è indicato come *hash length*. Ad esempio, se viene scelto un valore di hash length pari a 3, mediamente si prenderà un pacchetto su otto.

3.2.4 Dynamic hash sampling

L'accoppiata hash sampling e PNPM sembra essere efficace, ma soffre di un particolare difetto: la scelta del valore dell'hash length. Una decisione errata porterebbe ad avere o un numero molto basso di pacchetti selezionati, portando quindi a misure non significative, o un numero decisamente elevato, allungando i tempi per il processamento finale delle misure raccolte dai singoli apparati. Si è quindi pensato di creare un "hash length adattivo", il cui valore si adegui al rate dei pacchetti. A inizio test viene fissato un valore massimo, Max, di misure desiderate. Se tale valore viene raggiunto si incrementa il valore dell'hash length. L'incremento del valore di hash length renderà più difficile l'acquisizione di nuovi campioni, ma tale cambiamento si riflette anche suoi pacchetti già raccolti.

Infatti, avendo "allungato" l'hash length, non si è più certi che il confronto tra valore di riferimento ed hash sia ancora valido. E' quindi necessario effettuare dei controlli sui dati raccolti in precedenza per sapere quante nuove informazioni sono necessarie.

3.3 PNPM e sincronizzazione

Per quanto è stato detto in precedenza, PNPM ha necessità di avere gli apparati sincronizzati fra loro, per poter confrontare correttamente le misure legate al OWAD e OWAJ. Queste problematiche sono gestite da TIM, mentre nello sviluppo del software ci si è concentrati sul recupero e sulla gestione dei timestamp. Durante lo sviluppo del programma è stato infatti necessario individuare quale sia il clock corretto da usare poiché, all'interno dei sistemi Linux, ve ne sono diverse tipologie [15] e ognuno di essi gode di determinate proprietà, tali da rendere il loro uso idoneo solo in alcuni specifici contesti. La nostra analisi si concentra su due scelte, in quanto solo questi clock erano direttamente, o indirettamente, accessibili dai programmi eBPF.

- **CLOCK_MONOTONIC**: valore assoluto indicante la distanza temporale da un arbitrario evento passato, tipicamente associato all'istante di boot dell'apparato. Il suo valore non viene influenzato da fattori esterni e può solamente crescere nel tempo, eccezion fatta durante le operazioni di reboot dove ne viene fatto l'azzeramento. Poiché il singolo valore non ha un grande significato, in quanto non è associabile in alcun modo al tempo "reale", viene tipicamente usato per misurare il delta fra due eventi all'interno dello stesso sistema.

- **CLOCK_REALTIME**: fornisce la miglior approssimazione dell'istante di tempo rispetto la Unix Epoch ed è influenzato dai protocolli di sincronizzazione quali NTP e PTP. Tali modifiche possono comportare dei veri e propri "salti in avanti e indietro nel tempo", al fine di ottenere il valore più preciso possibile. Viene comunemente usato per confrontare le misure temporali acquisite da due o più apparati.

Risulta immediato capire quale sia stata la tipologia di clock usata nelle nostre analisi.
Verrà illustrato in seguito il metodo con cui tale valore è leggibile dai programmi eBPF e le problematiche ad esso associate. Non è stato invece oggetto di indagine la valutazione della precisione o dell'accuratezza dei timestamp ottenuti, in quanto la tematica è notevolmente complessa e articolata. Si rimanda quindi ai già numerosi studi presenti in letteratura [14], [16].

3.4 eBPF

3.4.1 BPF

Il Berkeley Packet Filter fu introdotto nel 1997, a partire dalla versione 2.1.75 del Linux kernel. Lo scopo principale per cui BPF nacque fu la necessità di eseguire in modo efficiente e veloce le operazioni di packet filtering lato kernel. L'esigenza di spostare l'elaborazione era forte, in quanto si evitava l'overhead delle copie di pacchetti non desiderati dal kernel space all'user space. Il funzionamento di BPF si basa sull'uso di una virtual machine lato kernel, la quale permette l'esecuzione di piccoli programmi, scritti usando il BPF bytecode

3.4.2 BPF virtual machine

La virtual machine è il componente chiave di BPF. E' dotata di registri e un set di istruzioni dedicati, con l'obiettivo specifico di massimizzare le prestazioni del packet filtering. Per poter eseguire il BPF bytecode questo deve essere calato nella virtual machine che, grazie al Just-In-Time(JIT) compiler, verrà successivamente tradotto nelle istruzioni native della macchina.
Poiché le operazioni legate all'analisi dei pacchetti sono relativamente semplici il BPF bytcode è composto solo da salti condizionali, operazioni aritmetiche, comparazioni e istruzioni per la lettura/scrittura di dati da/verso la memoria.

```
1. ldh[12]
2. jeq #0x800 jt 3 jf 4
3. ret #96
4. ret #0
```

Esempio BPF bytecode generato da un filtraggio basato sui pacchetti di tipo IPv4.

Questo codice è un semplice esempio del BPF bytecode. In particolare, il filtro come prima operazione si occupa di andare a caricare l'ethertype in un registro ed effettuare una comparazione con il valore 0x800, che rappresenta il valore IPv4 del campo ethertype. Se il risultato di tale operazione è positivo allora il pacchetto verrà spostato in user space, altrimenti verrà ignorato.

Come si può notare la scrittura diretta del BPF bytecode non è molto user friendly. Esistono fortunatamente alcune librerie, WinPcap per Windows e LibPcap per Linux, che permettono la generazione e la compilazione del BPF bytecode scrivendo un semplice programma C. Usando tali librerie è inoltre possibile caricare nella virtual machine il packet filter scritto.

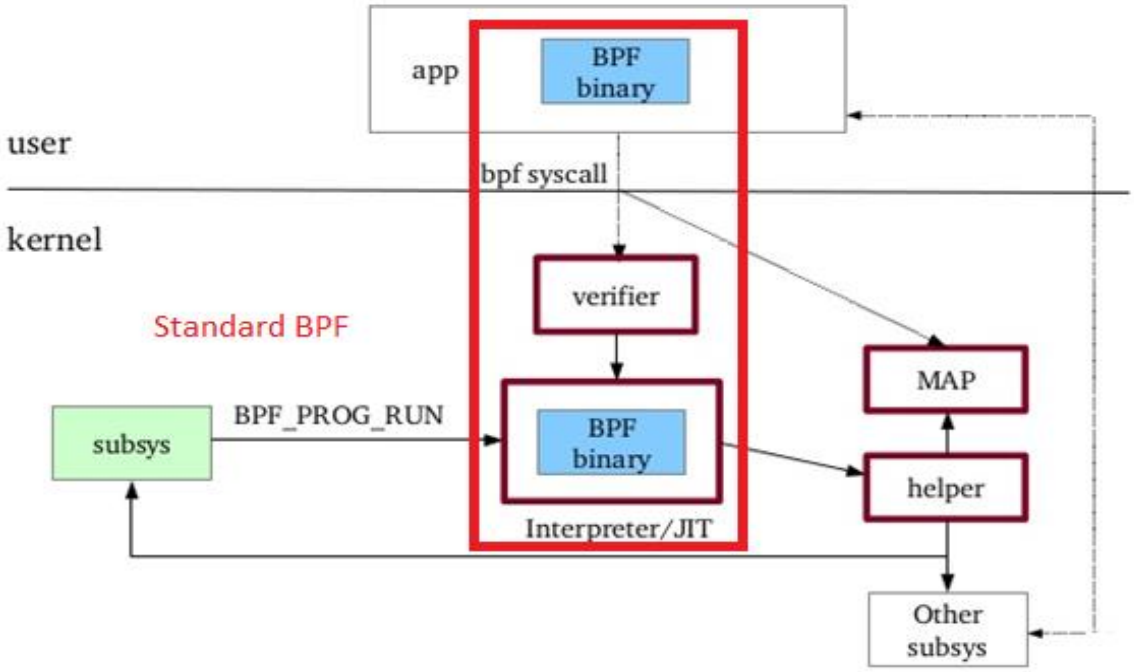


Figura 7 Interazione di BPF/ eBPF con il sistema – Rif. [17]

3.4.3 Sicurezza e limiti

Poiché i programmi BPF vengono eseguiti lato kernel è necessario garantire che non insorgano problemi, che potrebbero minare la sicurezza del sistema. Per queste ragioni è stato introdotto un componente nel kernel, il BPF Verifier, il cui compito è analizzare il codice scritto, controllando che il numero di istruzioni generate sia finito e che non vi siano accessi di memoria invalidi. In particolare vengono eseguiti i seguenti controlli:

- non vengano superate il numero di istruzioni massime
- non siano presenti dei loop
- le istruzioni di jump accedano a porzioni di memoria valide
- correttezza sintattica delle istruzioni e dei loro argomenti

Tutte questi controlli rendono la virtual machine un ambiente sicuro e, poiché non è possibile accedere indiscriminatamente a porzioni del kernel, isolato dal sistema.

BPF insns program (pre-3.15)	Extended BPF insns program
2 registers + stack 32-bit registers 4-byte load/store to stack 1-8 byte load from packet Conditional jump forward +, -, *, … instructions	10 registers + stack 64-bit registers 1-8 byte load/store to stack 1-8 byte load/store to packet Conditional jump fwd and backward Same + signed_shift + bswap

Figura 8 Confronto tra la virtual machine di BPF e quella di eBPF – Rif. [18]

3.4.4 eBPF

Negli ultimi anni si è assistito ad una evoluzione di BPF, non più limitato al packet filtering, ma permettendo all'interno della virtual machine lato kernel l'esecuzione di

codice generico, con le dovute limitazioni per motivi di sicurezza. eBPF ha introdotto diverse novità in uno scenario pressoché statico da 20 anni.

- diverse tipologie di programmi, da quelli orientati al processamento dei pacchetti, non solo filtraggio, ai programmi di tracing.
- le mappe, porzioni di memoria che possono essere condivise tra user e kernel space, o tra più programmi eBPF che vengono eseguiti in parallelo.
- funzioni helpers, speciali funzioni per poter interagire con il kernel e semplificare la scrittura dei programmi.

3.4.5 Tipologia di programmi eBPF

Vengono qui illustrati brevemente le varie tipologie di programmi eBPF:

- **SOCKET_FILTER**: possono attaccati ai socket ed eseguono le classiche operazioni di packet filtering, ritornando in user space i pacchetti che superano il filtraggio.
- **TRACING**: ogni volta che un evento del sistema viene generato si avvia un programma eBPF, che è stato associato a tale evento. Esistono diversi tipi di eventi a cui i programmi possono essere legati: kprobe, uprobe, tracepoint.
- **SCHED_CLS** e **SCHED_ACT**: i programmi eBPF sono agganciati al traffic control (TC), un componente del sistema di networking all"interno del kernel Linux. Questi programmi possono eseguire diversi azioni sul pacchetto, da classici drop e redirect a modifiche dei campi. Risultano quindi ideali per implementare delle funzionalità di rete.
- **XDP**: i programmi possono interagire con la tecnologia di Express Data Path (XDP). XDP permette infatti di processare i pacchetti prima che qualsiasi struttura dati del kernel venga allocata, quindi nel punto più basso nel networking stack. Questo permette di raggiungere velocità molto elevate.

3.4.6 Mappe eBPF

Le mappe rappresentano sicuramente una delle novità più significative all'interno di eBPF poiché permettono sia di introdurre il concetto di "stato" fra le diverse invocazioni di uno stesso programma, salvando ad esempio i valori ottenuti dall'ultima esecuzione e comparandoli con quelli attuali, sia per l'esportazione dei risultati lato user space. Nelle evoluzioni che eBPF ha avuto, e che continua ad avere, sono state aggiunte diverse tipologie di mappe, quali:

- **BPF_HASH**: permette la creazione di una hash map, una struttura dati "chiave/valore". Usando la chiave è quindi possibile recuperare direttamente il valore immagazzinato in precedenza. Risultano quindi molto adatte per operazioni di inserimento e successiva lettura.
- **BPF_ARRAY**: array indicizzato tramite l'uso di un intero. Anch'esso è ottimizzato per le operazioni di lettura e aggiornamento.
- **BPF_HISTOGRAM**: crea una mappa utile nel salvataggio di statistiche organizzate ad istogramma.

L'elenco sopra presentato è meramente indicativo, per una lista completa si faccia riferimento a [19].

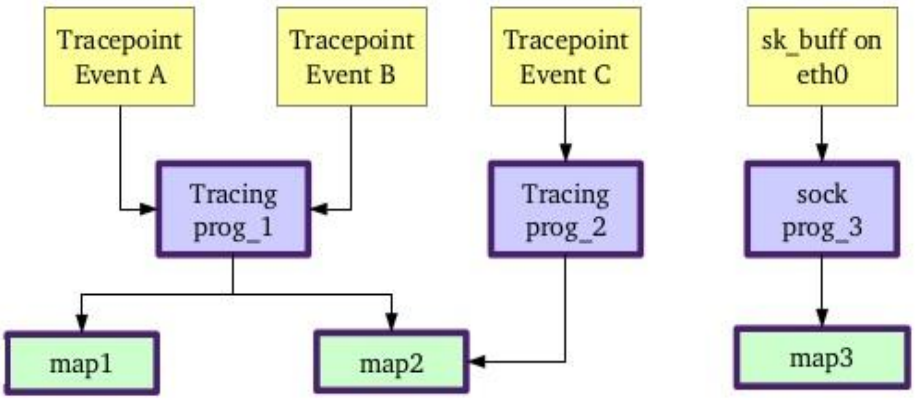


Figura 9 Possibili interazioni programma eBPF e mappe – Rif [17]

3.4.7 Metodi helpers

Per motivi di sicurezza, la virtual machine è totalmente isolata e non può richiamare funzioni esterne. Le funzioni helpers rappresentano quindi gli unici metodi per accedere alle funzionalità del sistema. Questi helpers sono integrati a livello kernel e ad ogni aggiornamento ne vengono introdotti di nuovi. Il loro uso è però attualmente limitato in base tipologia del programma eBPF che si sta realizzando.

- Map Lookup/Update/Delete
- Get ktime
- printk to trace buffer
- Get random number
- Get SMP processor ID
- Load/store n bytes in skb data
- Replace L3/L4 checksum of skb
- Name/UID/GID of current process
- Push/pop VLAN header
- Set/get tunnel key and options
- Tail call
- Read/write perf event ring buffer
- Redirect/clone to other net_device
- Get routing realm
- Calculate checksum diff over memory buffer
- Change protocol of skb
- Change type of skb (local/broadcast/…)
- Check for cgroup membership
- Access skb->hash or mark invalid
- Trim tail of skb
- Make skb linear

Figura 10 Alcuni metodi helpers - Rif. [20]

3.5 IOVisor e BCC

IOVisor è un progetto open source nato con l'obiettivo di sviluppare e condividere nuove funzionalità di rete e di IO. Negli anni IOVisor ha contribuito al miglioramento di eBPF, concentrandosi sia sulla flessibilità che sulle performance in diversi tipologie di ambiti, quali networking e tracing.

Come è già stato evidenziato in questo capitolo, la scrittura, la validazione e la compilazione dei programmi eBPF sono compiti gravosi e dispendiosi in termini di tempo. BPF Compiler Collection [21] (BCC) è un toolkit sviluppato da IO Visor che mira

ad assolvere tutte queste funzionalità. Il fine ultimo è quello di consentire al programmatore di usare linguaggi di più alto livello per la realizzazione dei propri tool. BCC fungerà da front-end, automatizzando il processo di generazione del bytecode e il relativo inserimento nella virtual machine.

Le principali caratteristiche di BCC sono:

- End-to-end BPF workflow
- Uso di un linuaggio C "ristretto" per la parte back-end
- L'integrazione di llvm-bpf backend e uso del "JIT compiler"
- Caricamento dinamico dei programmi "JITted"
- Supporto di eBPF ai "kernel hooks": socket, TC, kprobe,…

BCC fornisce quindi un ulteriore livello di astrazione, andando ad occuparsi della compilazione del codice C in bytecode, sfruttando "clang/llvm" per poi usare la system call "bpf()" per l'iniezione del programma lato kernel. Tramite Python, linguaggio principale usato per la scrittura del front-end, sarà poi possibile interagire con le mappe, per leggere i dati salvati.

4 Architettura IOVisor-PNPM

4.1 Introduzione architettura

Come è stato accennato in precedenza IOVisor-PNPM è suddiviso in una sezione in user space, con cui si può interagire tramite un'interfaccia RESTfull API, mentre lato kernel esistono tre tipologie di programmi eBPF, ognuno dei quali ottiene le misure relative a packet loss, delay e jitter usando tecniche diverse. Come verrà illustrato in questo capitolo, tutti i programmi eBPF hanno una parte in comune, inerente al riconoscimento dei flussi e al salvataggio delle misure. I programmi si differenziano invece per:

- l'uso, o meno, di tecniche di packet sampling, usando N pacchetti degli M ricevuti per calcolare le misure desiderate.
- la tipologia delle misure. Si possono usare misure medie, calcolate sull'unità "blocco", di durata MP, o misure puntali, ovvero valori associati al specifico pacchetto.

Tramite l'interfaccia REST è possibile indicare quale tipologia di programma eBPF calare nella virtual machine e configurarla usando opportuni parametri.

4.2 Architettura

Verrà ora presentata l'architettura del software sviluppato. In questa prima parte ci si focalizzerà nell'evidenziare, ad alto livello, i componenti, suddivisi fra kernel e user, e le loro funzionalità, mentre nella seconda parte si andranno ad analizzare più in dettaglio le operazioni svolte da ogni componente. I dettagli implementativi sono invece lasciati al capitolo cinque.

4.2.1 Componenti kernel space

- **Interfaccia di rete**: l'interfaccia di rete rappresenta il punto di ingresso dei dati nel nostro sistema. Il processamento dei pacchetti inizierà nel momento in cui i

suddetti vengono spostati dall'interfaccia al kernel, in particolare quando raggiungono la funzione del kernel monitorata.

- **Hook point**: punto di "aggancio" dei programmi eBPF che operano nella modalità tracing, usufruendo del meccanismo delle kprobe. Questo implica che ogni qualvolta che verrà richiamata la funzione del kernel questa causerà l'esecuzione del programma, calato in precedenza nella virtual machine. Nel contesto considerato la funzione è quella dedita alla gestione dei pacchetti ricevuti.
- **Programma eBPF**: il programma di tracing potrà accedere agli argomenti della funzione che ne ha causato l'esecuzione, ottenendo così le informazioni associate ad ogni pacchetto (campi, timestamp, etc). Tali informazioni saranno manipolate e salvate successivamente nelle mappe.
- **Mappe**: costituiscono il punto di contatto tra il kernel e l'user space. Saranno lette e scritte da ambo i contesti, in fase di configurazione e in fase di recupero dei risultati associati al singolo blocco.

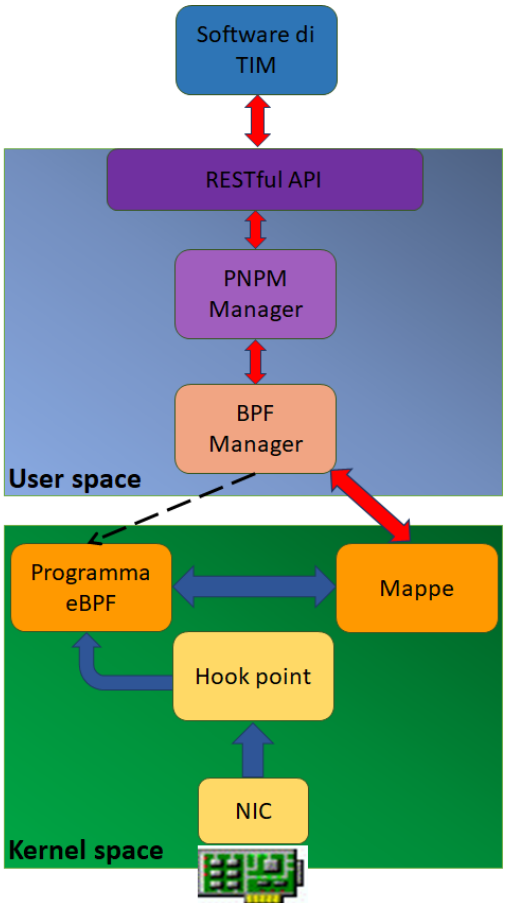


Figura 10 Diagramma architetturale di IOVisor-PNPM

4.2.2 Componenti user space

- **Software di TIM**: programma generico dedicato alla configurazione di IOVisor-PNPM e alla raccolta delle misure. Queste ultime verranno successivamente confrontate fra loro, per ottenere le misure one-way e two-way.
- **Interfaccia REST**: porzione del programma dedicata alla ricezione dei parametri di configurazione dei programmi eBPF e all'esportazione dei risultati ottenuti nella fase di test.
- **PNPM Manager**: interagisce con il BPF Manager, e con il software di TIM attraverso l'interfaccia REST. Implementa la logica PNPM e coordina le operazioni fra gli altri componenti.
- **BPF Manager**: gestore dei programmi eBPF che fornisce le funzionalità per poter lavorare con i suddetti programmi. Tali metodi riguardano la creazione e la configurazione del programma, l'inizializzazione delle mappe e le ttura dati.

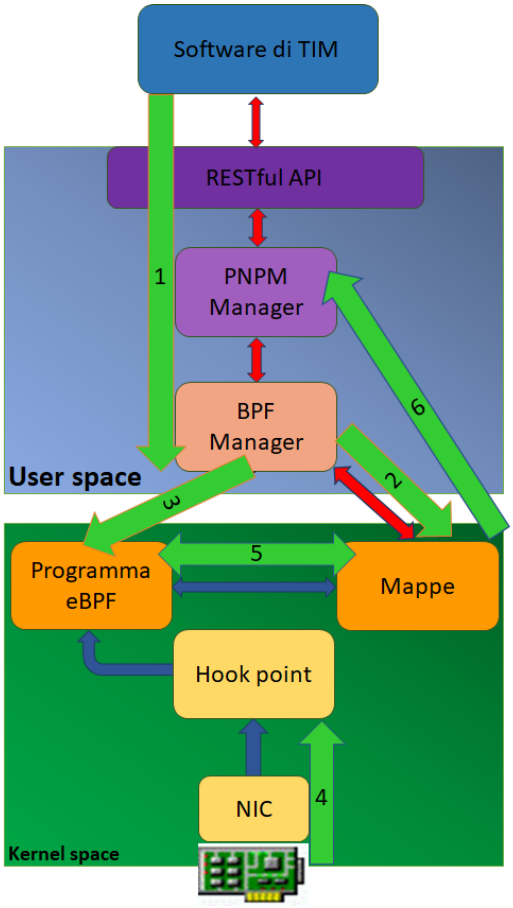


Figura 11 Workflow in IOVisor-PNPM

4.3 Workflow

Si metteranno ora in evidenza le macro fasi che portano alla corretta inizializzazione di un programma eBPF di tracing e alla successiva lettura dei dati.

1. IOVisor-PNPM riceve un file di configurazione nel quale vengono fornite informazioni riguardo la tipologia di programma eBPF da usare, i parametri legati alla sessione di test PNPM e la descrizione dei filtri per identificare i pacchetti. Il PNPM Manager, interagendo con il BPF Manager, prepara quindi il programma e le strutture dati per un successivo avvio.
2. Il BPF Manager associa ad alcune costanti i valori passati in fase di configurazione e interagisce con le mappe per l'inizializzazione dei filtri. Se tutte le operazioni sono andate a buon fine IOVisor-PNPM avvisa che il porgramma è stato correttamente configurato.
3. Quando viene ricevuto il comando di avvio, IOVisor-PNPM provvede ad iniettare il programma eBPF, configurato in precedenza, nella virtual machine e avvia un processo parallelo per sua gestione.
4. I pacchetti ricevuti dalla NIC si muovono risalendo il networking stack per giungere alla funzione del kernel da noi osservata, l'hook point. Questo causa l'avvio del programma eBPF.
5. Il programma eBPF esamina le informazioni del pacchetto, cerca di associarlo a qualche filtro e, se l'operazione ha esito positivo, aggiorna le strutture dati nelle mappe.
6. Il processo in user space dedicato alla gestione del test PNPM legge periodicamente dalle mappe i valori salvati e li riorganizza in apposite strutture dati lato user space.

4.4 Analisi componenti

Di seguito si forniscono maggiori dettagli rispetto alla descrizione funzionale dei componenti appena visti.

4.4.1 Interfacce di rete

La funzione del kernel viene richiamata, escludendo i casi di pacchetti errati o droppati, ogni qualvolta che un pacchetto viene ricevuto su una qualsiasi interfaccia e viene mandato al networking stack. Questo implica che se all'interno del nostro sistema utilizziamo dei meccanismi di routing sui pacchetti, i programmi di tracing opereranno ogni volta che il pacchetto verrà identificato come se fosse stato ricevuto da una certa interfaccia. E' possibile in fase di configurazione andare a indicare su quali interfacce si desiderare lavorare. Questo non impedirà al programma eBPF di essere richiamato ma eviterà, per lo meno, elaborazioni superflue.

4.4.2 Hook Point

E' stato detto in precedenza che esistono diverse tipologie di programmi eBPF. Sebbene sia possibile classificare i programmi sviluppati nella categoria del monitoraggio, tali programmi riguardano sostanzialmente la ricezione dei pacchetti. La prima scelta valutata fu quella delle modalità SCHED_CLS o SCHED_ACT in quanto tali programmi eBPF hanno come punto di aggancio il TC, componente del sottosistema di networking all'interno del kernel Linux. Tale componente è dedicato esclusivamente al processamento dei pacchetti, permettendone fra l'altro la modifica, il redirect, il drop etc.

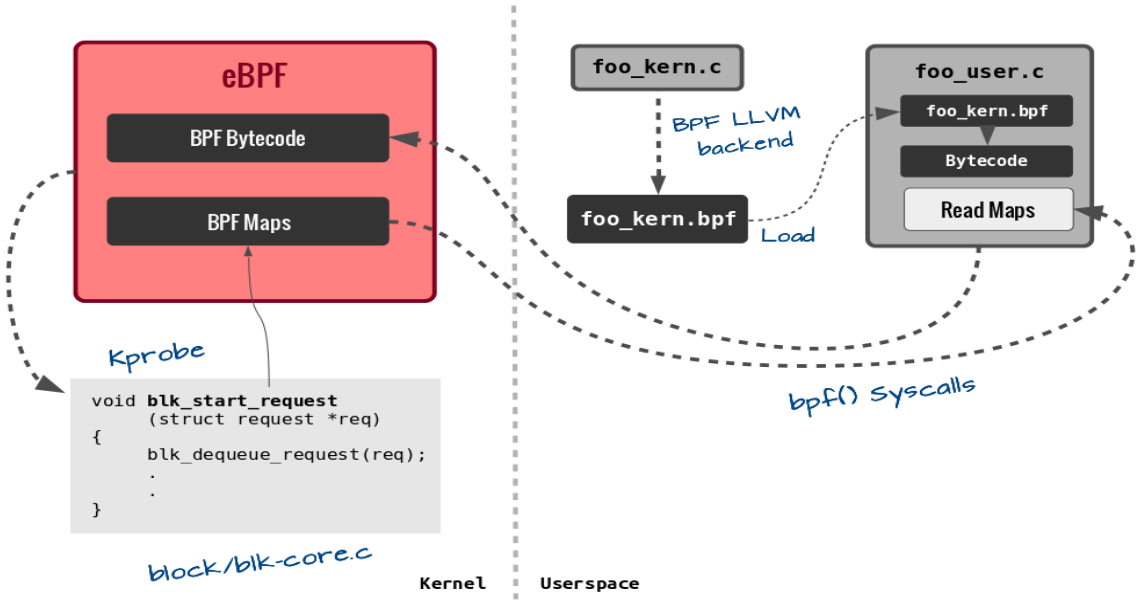


Figura 12 Esempio funzionamento kprobe. - Rif. [22]

Come verrà illustrato nel capitolo successivo, l'implementazione attuale di un programma eBPF legato al TC usa una struttura dati ridotta rispetto a quella standard usata nel kernel. Tra le informazioni mancanti vi è il timestamp, dato fondamentale per il monitoraggio secondo PNPM. Per tali motivazioni è quindi stata scelta la tipologia TRACING.

Da notare che la funzione individuata come kprobe viene considerata come il punto di inizio del TC.

4.4.3 Programmi eBPF

Tutte i programmi eBPF scritti seguono due step fondamentali:

1. **Riconoscimento del flusso**: vengono letti alcuni campi dei pacchetti in arrivo al kernel, con lo scopo di controllare se essi siano associati ai flussi da monitorare. In caso positivo l'elaborazione prosegue con lo step due altrimenti l'esecuzione del programma termina, per passare all'analisi del pacchetto successivo. In questa fase viene anche letto il valore del campo DSCP, usato per marcare i pacchetti.

2. **Elaborazione dei dati**: si inizializzano/si aggiornano le strutture dati dedicate al monitoraggio delle prestazioni di rete. Tali dati vengono salvati nelle mappe, per pos essere ciclicamente letti dal programma di gestione in user space.

4.4.3.1 Riconoscimento flussi

Volendo fornire contemporaneamente una buona flessibilità e una discreta complessità nella descrizione dei filtri, si è deciso di realizzare una struttura dati denominata *struct filter*, composta dai parametri caratterizzanti un flusso da monitorare. La scelta dei campi è ricaduta sulla quintupla standard per l'identificazione di un flusso TCP/IP. Tali filtri sono salvati nelle mappe nella fase di inizializzazione del programma, cosicché, per ogni pacchetto in arrivo, sia possibile comprendere se esso sia da processare. In particolare,

con la metodologia descritta in seguito, è possibile far uso nei filtri delle wildcard[1], permettendone una scrittura più generica.

Un'altra possibilità per la realizzazione dei filtri è la conversione dei parametri identificanti il flusso come stringa, composta da "if" a condizioni multiple; ogni "if complesso" identifica quindi un flusso da monitorare. Tale stringa verrebbe poi iniettata in fase di configurazione nel programma C-like. Il numero di filtri così configurabili dipende però dalla complessità del programma, poiché le catene di "if complessi" rientrano nel limite massimo di istruzioni generate. Poiché il numero totale di filtri realizzabili con questo approccio era basso rispetto a quanto desiderava TIM si è deciso di usare il primo approccio, basato sulle mappe.

4.4.3.2 Elaborazione dati

La parte di processamento dei dati è suddivisa a sue volta in due sottosezioni:

- **Packet loss**: per individuare eventuali perdite è necessario conteggiare continuamente i pacchetti ricevuti, anche se questi non verranno poi usati come campioni per ulteriori misure, legate al delay o al jitter.

- **Delay e jitter**: la raccolta dei timestamp dei pacchetti, usati per calcolare tali misure, dipende dalla modalità di tracing selezionata, basata o meno sul principio del packet sampling.

Le misure di delay e jitter possono essere medie, associate all'unita blocco, o puntuali per i singoli pacchetti, a secondo del programma scelto.

[1] Le wildcard sono dei "caratteri jolly", che possono assumere qualsiasi valore.

Vedremo ora in cosa si contraddistinguono le tre tipologie di programmi di tracing.

4.4.3.3 Programma uno – Analisi di tutti i pacchetti

Questo programma esegue gli step base di riconoscimento ed elaborazione dei dati. Le misure ottenute sono medie, associate al blocco.

4.4.3.4 Programma due – Analisi di N pacchetti, modalità "fixed hash"

Sfruttando il meccanismo dell'hash sampling si usa un sottoinsieme di N pacchetti, rispetto gli M ricevuti, per aggiornare le statistiche. In particolare, si andranno a selezionare solo quei pacchetti il cui hash è pari ad un certo valore, passato in fase di inizializzazione del programma. Anche la funzione di hash da usare è scelta in fase di configurazione. L'hash length indica invece il numero di bit da considerare per il confronto. In questa modalità il numero dei pacchetti utilizzati per l'aggiornamento delle misure non è predicibile e dipende solamente dal valore scelto e dall'hash length.

Le misure ottenute in questa modalità sono ancora medie ora, però, rispetto ai pacchetti campionati, non rispetto al totale.

Si ricorda che il numero delle misure ottenute non è ovviamente noto nell'istante di configurazione ed è solamente stimabile: esso dipende dal valore di match, dal hash length e dai pacchetti in arrivo.

4.4.3.5 Programma tre – Analisi di [Max/2, Max] pacchetti, modalità "dynamic hash"

Evoluzione del secondo programma, dove il valore di "hash length" può crescere nel tempo. Infatti se il numero di pacchetti selezionati è troppo grande, raggiungendo un valore soglia pari a Max, il numero di bit usati nella fase di matching viene incrementato di uno. Questo implica che sarà più difficile trovare un pacchetto il cui valore di hash sia uguale a quello di riferimento. Si sottolinea inoltre che i valori vecchi potrebbero non essere più validi, poiché è cambiato il valore di riferimento. E' quindi necessario effettuare un nuovo controllo sui valori già accumulati effettuando l'operazione di matching.

Le misure salvate sono sia associate al blocco, packet loss, sia ai pacchetti, come il timestamp. L'uso di misure puntuali permetterà, sfruttando l'hash come identificativo, un confronto diretto fra i valori raccolti da più apparati. L'eventuale gestione di collisioni è demandata al software di TIM.

Come si può già notare dal funzionamento generale di questo programma di tracing, vi è la necessità di usare un ciclo per ripetere il controllo sui vecchi valori accumulati, cercando quelli da scartare. Come è stato accennato, i loop nei programmi eBPF sono proibiti. Vedremo nel prossimo capitolo quali soluzioni sono state adottate e se esse si saranno dimostrate valide.

4.4.4 BPF Manager

Il BPF Manager ha il ruolo di gestore dei programmi BPF e permette di eseguire diverse operazioni:

- configurazione del programma di tracing
- attach/detach del programma eBPF all/dall'hook point.
- lettura delle mappe, ove saranno contenute le misure salvate.

Si evidenzia che le operazioni svolte dal BPF Manager non hanno al loro interno alcuna logica, si limitano ad effettuare le operazioni descritte. L'algoritmo che determina quali e quando usare tali funzioni è situato nel PNPM Manager.

4.4.5 PNPM Manager

Il PNPM Manager funge da intermediario fra l'interfaccia REST e il BPF Manager, permettendo una migliore suddivisione dei compiti. Il PNPM nella fase di configurazione si limita a richiamare il BPF Manager, mentre nelle fasi di monitoraggio coordina le operazioni di lettura affinché queste vengano svolte secondo le giuste tempistiche.

I dati letti durante la fase di test saranno resi disponibili all'esterno grazie ad apposite chiamate dell'interfaccia REST. Terminato il periodo di test il programma di tracing viene

eliminato, i risultati rimangono invece disponibili finché non si avvia una nuova sessione di test o non si elimina il programma.

5 Implementazione IOVisor-PNPM

In questo capitolo verrà spiegato in modo più dettagliato l'implementazione dei componenti, sia quelli in kernel space che quelli in user space, di IOVisor-PNPM. Viene fatta una breve descrizione della struttura dati del kernel contenente tutte le informazioni associate al pacchetto, *sk_buff*, e si spiegherà nel dettaglio l'accesso al timestamp del pacchetto. Si procederà poi ad esaminare i programmi eBPF e infine si vedranno i componenti di gestione in user space.

5.1 Sk_buff

La struttura dati del kernel *sk_buff* contiene tutte le informazioni utili all'analisi dei pacchetti fra cui:

- il puntatore per scorrere i campi del pacchetto,
- le informazioni relative all'interfaccia su cui il pacchetto è stato ricevuto
- timestamp

Questa è anche la struttura dati che la funzione *__netif_receive_skb_core* riceve come argomento. Ogni qualvolta che il programma eBPF viene avviato potrà leggere le informazioni relative al nuovo pacchetto esaminando tale parametro.

La lettura dei campi del pacchetto è possibile tramite l'uso dell'aritmetica dei puntatori del C. Di base il puntatore *data* punta ai campi del livello data-link, ma aggiornandone il valore è possibile spostarsi al livello rete, e successivi. Per ognuno dei livelli del modello ISO/OSI esistono delle idonee strutture dati a livello kernel con cui è possibile accedere ai dati in modo agevole.

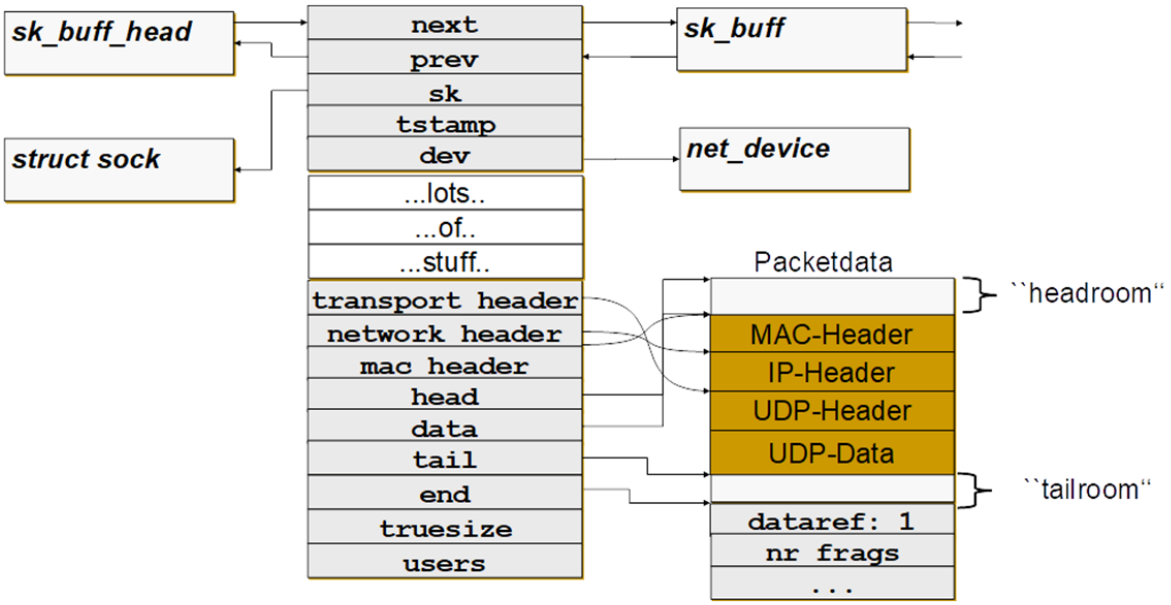


Figura 13 Struttura dati sk_buff. – Rif. [23]

```
1.  struct udphdr
2.  {
3.      __be16  source;
4.      __be16  dest;
5.      __be16  len;
6.      __sum16 check;
7.  };
8.
9.  ...
10.
11. struct udphdr* udphdr = data;
12. flow->sport = bpf_ntohs(udphdr->source);
```

Esempio di gestione dei pacchetti.

Altre tipologie di programmi, quali SCHED_CLS/SCHED_ACT e XDP sfruttano un'altra struttura dati, denominata *__sk_buff*. Tale struttura dati è specifica per i programmi eBPF e presenta diverse differenze rispetto a *sk_buff* standard. La struttura *__sk_buff* appare molto più compatta rispetto quella standard e compaiono alcuni nuovi campi, ideati per risolvere le problematiche legate allo sviluppo delle funzionalità di rete.

Fra i campi mancati vi è il timestamp, motivo per cui si è scelto di usare la modalità TRACING per lo sviluppo dei nostri programmi.

5.2 Acquisizione timestamp

Nel capitolo tre sono stati descritti i punti chiave del funzionamento di PNPM, fra cui la raccolta dei timestamp e il successivo confronto delle misure ottenute. E' anche stata esplicitata la necessità di accedere ad un timestamp il cui valore sia confrontabile con quello di altri apparati.

BCC fornisce un helper *bpf_ktime_get_ns()* con il quale è possibile ricavare i valori, in nanosecondi, dal CLOCK_MONOTONIC, mentre non è previsto un supporto per il CLOCK_REAL_TIME. Ricordiamo che nei programmi eBPF non è possibile richiamare le funzioni del kernel, eccezion fatta per i metodi helper. L'unica possibilità resta quindi di far generare al sistema, per tutti i pacchetti in arrivo, il timestamp di modo che sia poi leggibile dalla struttura *sk_buff*. Non vi è però un metodo diretto con cui indicare al sistema di generare i timestamp. E' quindi necessario creare un socket e usare le *setsocketoption*.

Questo approccio appare molto conveniente, perché garantisce la generazione di un valore temporale direttamente dal sistema senza doversi preoccupare di ulteriori dettagli, ma presenta due difetti:

- la generazione dei timestamp è un'operazione pesante e questo inciderà sulle prestazioni raggiungibili dai programmi di tracing.
- non è possibile decidere quali pacchetti necessitano il timestamp o no. Ciò sarebbe stato molto utile in abbinamento alle tecniche di hash sampling perché permetterebbe di investire meglio le risorse sui soli pacchetti di nostro interesse.

Si sottolinea infine che le problematiche legate al timestamp sono comuni a tutti i programmi eBPF e possono complicare la gestione o impedire del tutto la creazione di alcuni tools.

Si descriveranno ora più nel dettaglio il funzionamento dei tre programmi di tracing. Tali programmi sono stati scritti in un linguaggio C "ristretto", in quanto non è possibile far uso dei loop, e sfruttando gli helper messi a disposizione del framework BCC. Il codice finale viene trattato come stringa, per poi essere passato all'oggetto BPF, esportato da BCC. Il costruttore richiama quindi il *BPF Verifier*, il quale si occupa di effettuare alcuni controlli sintattici e verifica che non vengano violati i vincoli su cui la virtual machine si basa.

5.3 Programma uno - Analisi di tutti i pacchetti

5.3.1 Controllo interfacce di rete

La prima operazione svolta è il controllo che il pacchetto arrivi da un'interfaccia su cui stiamo effettuando il monitoraggio, altrimenti l'esecuzione del programma terminerebbe. L'operazione di riconoscimento dell'interfaccia viene effettuata esaminando *sk_buff*, il quale contiene un riferimento alla struttura dati *net_device*. *Net_device* contiene tutte le informazioni relative alle interfacce, fra cui il nome e il numero identificativo. In fase di configurazione del programma viene fatta l'iniezione del codice relativo al controllo degli identificativi validi.

5.3.2 Riconoscimento dei flussi

L'approccio scelto per identificare i flussi è quello basato sull'uso della struttura dati *struct filter* e l'uso delle mappe. Ricevuta la descrizione del filtro, tramite il file di configurazione, questa viene scomposta nei campi singoli che lo identificano. Viene quindi creata una prima istanza di *struct filter* e viene inizializzata con il valore del primo campo del filtro, mentre gli altri campo vengono settati a zero. Tale struttura dati così composta viene salvata nella mappa di tipo hash e rappresenta una parte del filtro finale. Vengono quindi fatti altri inserimenti, e per ogni inserimento si aggiunge il valore di un nuovo campo. Questo approccio permette di gestire sia filtri con condizioni lasche sia casi in cui un flusso sia identificato univocamente tramite la quintupla standard.

5.3.2.1 Strutture dati

Si usa una mappa di tipologia hash, che permette un'associazione fra i dati di tipo *struct filter*, e un valore numerico. Il numero ha due funzioni:

- distinguere le entries relative ai pezzi del filtro da quelle rappresentanti il filtro stesso.
- svolge il ruolo di identificativo del filtro, in particolare si usa il valore con cui é stato salvato lato user space. Questo permette lato programma eBPF di avere strutture dati più compatte, poiché usano solo questo valore numerico invece di tutti i campi, e lato user space si ha la possibilità di avere accesso diretto alle misure usando l'identificativo.

Il campo FILTERS è una costante e rappresenta il numero di entries che si desiderano nella mappa. Il valore di tale costante è calcolato in fase di configurazione dei programmi ed è settato prima che si ricavi il corrispondente BPF bytecode.

```
1. struct filter
2. {
3.     u32 sip;
4.     u32 dip;
5.     u8 proto;
6.     u16 sport;
7.     u16 dport;
8. };
9.
10. ...
11.
12. BPF_HASH(filter_pieces, struct filter, int, FILTERS*5);
13.
14. ...
15.
16. flow->sip = bpf_ntohl(iph->saddr);
17. value = filter_pieces.lookup(flow);
18. if(value == NULL)
19.     flow->sip = 0;
20. else if(*value >= 0)
21.     goto FIND;
```

Riconoscimento dei flussi.

5.3.2.2 Algoritmo

Come si può vedere dal codice, durante l'analisi dei campi, l'istanza della struttura dati *filter* viene aggiornata in base all'esito dell'accesso della mappa. Se l'esito dell'accesso è positivo significa che è stata trovata un'istanza con i medesimi valori. Ciò implica che l'ultimo campo aggiunto è corretto, quindi va salvato. In caso contrario il valore dell'istanza viene riportato a zero. Un ulteriore controllo viene fatto sul valore ritornato dal filtro: se il numero è uguale o maggiore di zero significa che è stato individuato il filtro completo, altrimenti si è trovato un pezzo del filtro. Ripetuta tale procedura per un numero massimo di volte pari al numero di campi che compongono il filtro, i valori dell'istanza del filtro saranno stati completamente determinati.

La flessibilità che questo metodo garantisce viene pagata dai ripetuti accessi necessari alle mappe. In particolare il numero di accessi è direttamente proporzionale al numero di campi che compongo il filtro mentre è indipendente dal numero dei filtri salvati nella mappa.

5.3.3 Aggiornamento misure

5.3.3.1 Strutture dati

Vengono usate due mappe, aventi come chiave la coppia *filterd_id* e *netif_id*, mentre come valore associato la struttura dati *struct measures*, contenente tutti i campi utili per le misure. Si fa notare che il numero di contatori è quindi pari a due, moltiplicato per il numero dei filtri e per il numero delle interfacce. Questo valore è uguale a quello indicato nella descrizione teorica di PNPM.

```
1. struct marked_flow
2. {
3.     u16 netif_index;
4.     u16 id;
5. };
6.
7. struct measures
8. {
```

```
9.      u32 count_pkts_measure;
10.     u64 sum_quot;
11.     u64 sum_mod;
12.     u64 first_tstamp;
13.     u64 last_tstamp;
14. };
15.
16. BPF_HASH(flow_measures_color0, struct marked_flow, struct measures, F
    ILTERS);
```

Dettagli delle strutture per il salvataggio delle misure medie associati ai filtri.

5.3.3.2 Algoritmo

Se il flusso è stato riconosciuto si userà il suo identificativo e quello dell'interfaccia per suddividere le informazioni relative al conteggio dei pacchetti, del ritardo etc. Ciò implica che uno stesso flusso che passa attraverso due o più interfacce avrà misure diverse. Analoghe considerazioni valgono ovviamente per flussi diversi sulla stessa interfaccia.

Si evidenzia inoltre la necessità di avere due mappe, ognuna per le due possibili marcature del pacchetto durante la fase di testing. Questo approccio permette di gestire correttamente eventuali pacchetti, marcati con il vecchio valore, ritardatari.

```
1. if(mark == MARK0)
2.     mes = flow_measures_color0.lookup_or_init(&mf, &m);
3. else
4.     mes = flow_measures_color1.lookup_or_init(&mf, &m);
```

Salvataggio misure nelle mappe.

Grazie all'helper di BCC *lookup_or_init* è possibile svolgere più funzioni con la medesima istruzione:

- se nella mappa non vi è ancora una chiave avente coppia *filter_id*, *netif_id* allora significa che si è identificato il primo pacchetto di un nuovo flusso. E' quindi necessario inizializzare a zero tutte le variabili usate per le misure.
- se nella mappa è già presenta una chiave con quei valori allora vengono restituite le informazioni associate a quella chiave, permettendo così l'aggiornamento dei dati.

L'aggiornamento delle misure risulta semplice, è necessario prestare solamente attenzione alla corretta gestione dei timestamp. Essendo il valore, indicante il tempo trascorso in nanosecondi dal 1° Gennaio 1970, molto grande e dovendo farne la sommatoria, necessaria per poi calcolare il timestamp medio, si possono presentare dei casi di overflow. Poiché in Python, linguaggio usato per scrivere i componenti in user space, non vi è ancora un supporto per le variabili a 128bit si è pensato di dividere ogni timestamp per un numero molto elevato (es. un miliardo) e salvare in due variabili il quoziente e il resto. I nuovi valori ottenuti vengono aggiunti a quelli precedenti, permettendo poi lato user space di ottenere nuovamente la sommatoria e, conseguentemente, la media.

Le misure ottenute dal programma uno sono:

- conteggio dei pacchetti rilevati
- sommatorie dei resti e dei quozienti, dai quali si ricaverà il timestamp medio
- primo timestamp per un pacchetto appartenente ad un certo blocco di un certo flusso
- ultimo timestamp, valore aggiornato per ogni nuovo pacchetto rilevato

Questa metodologia di calcolo ovviamente introduce un errore dovuto all' approssimazione del resto, ma finché non vi sarà supporto per le variabili a 128bit è l'unica soluzione attuabile a fronte del problema dell'overflow.

5.4 Programma due - Analisi di N pacchetti, modalità "fixed hash"

Nella seconda modalità di tracing vengono usati solo una parte dei pacchetti per aggiornare le statistiche. Per selezionare i pacchetti si userà la tecnica di hash sampling, le cui funzionalità sono già state spiegate nel terzo capitolo. Tale step di campionamento è quindi posto fra il riconoscimento dei flussi e l'aggiornamento delle misure.

5.4.1 Scelta della funzione di hash

In fase di configurazione del programma è possibile indicare quali funzioni di hash si vuole usare per effettuare il campionamento dei pacchetti. Le funzioni attualmente supportate sono:

- IPSX
- BOB

Si ricorda che i programmi eBPF non possono sfruttare i metodi esposti da alcuna libreria o usare funzioni del kernel, eccetto gli helper messi a disposizione di BCC. Questo implica la riscrittura completa del codice della funzione di hash nel programma. Tale operazione può causare delle problematiche a causa del limite delle istruzioni, attualmente imposto a 4096 per ragioni di sicurezza.

Per non incorrere in tali problemi, e per rendere il codice più leggibile e pulito, si è pensato di sfruttare la gestione a stringa del programma eBPF, inserendo solo in fase di configurazione la funzione necessaria. Il codice delle funzioni di hash è disponibile nella documentazione di PSAMP [10].

5.4.2 Algoritmo

Durante l'analisi del pacchetto in una nuova struttura dati, *struct hash_input*, vengono salvati i valori usati dalla funzione di hash. *Struct hash_input* contiene un vettore di 5 elementi, che saranno processati per ottenere l'hash. A seconda del metodo scelto possiamo avere un hash a 32 o 16bit.

Ottenuto quindi l'hash associato al singolo pacchetto lo si confronta con il valore di riferimento. Se l'esito del confronto è positivo il pacchetto verrà usato per l'aggiornamento delle misure, altrimenti si proseguirà con l'elaborazione.

```
1. input->input[0] = (u32)( (bpf_ntohs(iph->id) << 16) + bpf_ntohs(iph-
   >frag_off) );
2. input->input[1] = bpf_ntohl(iph->saddr);
```

```
3. input->input[2] = bpf_ntohl(iph->daddr);
4. input->input[4] = (u32)bpf_ntohs(iph->tot_len);
5.
6. input->input[3] = bpf_ntohl(tcphdr->seq);
7. // oppure
8. input->input[3] = (u32)( (bpf_ntohs(udphdr-
   >len) << 16) + bpf_ntohs(udphdr->check) );
```

Salvataggio dati per il calcolo dell'hash

Le misure ottenute sono analoghe a quelle del programma uno, tranne per alcuni dettagli:

- i valori medi non sono più calcolati su tutti i pacchetti ricevuti, bensì sul numero di pacchetti selezionati.
- il primo timestamp è sempre riferito all'inizio del blocco mentre l'ultimo sarà riferito all'ultimo pacchetto scelto per l'aggiornamento delle misure.

Si può facilmente notare come i programmi uno e due hanno un comportamento molto simile, ma, secondo le ipotesi iniziali, l'uso dell'hash dovrebbe aumentare le prestazioni poiché si vanno ad esaminare un numero inferiore di timestamp. Vedremo dai risultati dei test di carico se tali ipotesi troveranno conferma.

5.5 Programma tre – Analisi di N pacchetti compresi fra Max/2 e Max, modalità "dynamic hash"

Facendo una panoramica dei programmi di tracing, la terza ed ultima tipologia era stata introdotta come un'evoluzione della seconda. Erano anche state accennate le problematiche legate a tale soluzione poiché il controllo sui vecchi valori accumulati, per verificarne la validità a fronte del nuovo valore di riferimento, implica l'uso di ciclo, costrutto proibito in BPF.

Si esporranno brevemente due soluzioni che sono state implementate ma che sono state poi abbandonate. La terza soluzione invece è una modifica parziale dell'idea originaria, cercando di sfruttare al meglio quanto BPF mette a disposizione.

5.5.1 Versione "looped" kernel space

In BPF, in realtà, i loop sono parzialmente supportati. E' possibile infatti scrivere alcuni loop se i valori estremi su cui si effettua l'iterazione sono delle costanti. Anche in questo caso è comunque possibile che il loop venga rilevato come "pericoloso" e quindi la generazione del bytecode fallisca. Nel nostro particolare caso il "BPF Verifier" mal tollerava l'uso dei loop associato agli accessi alle mappe. Per poter quindi effettuare un inserimento di un nuovo valore valido era necessario compiere un primo ciclo, per sovrascrivere una vecchia entries non più valida, e un secondo ciclo per aggiornare il contatore delle misure valide salvate.

Risulta evidente che il codice così scritto sia mal ottimizzato. Si è quindi pensato di abbandonare definitivamente l'uso dei loop in BPF, per evitare altre problematiche.

5.5.2 Versione "looped" user space

La seconda soluzione si basa sull'idea di lasciare al programma BPF solo il compito di effettuare gli inserimenti dei valori nelle mappe, mentre lato user space viene effettuato un loop per eliminare i valori non più validi. Il controllo dei vecchi valori presenta però alcune difficolta:

- deve essere effettuato quando si è raggiunto il numero massimo di misure possibili.
- per evitare di iterare sull'intera mappa dove vengono salvate le misure puntuali si ha la necessità, in user space, di conoscere per quale flusso sono state raccolte le informazioni sui pacchetti e quale è il nuovo valore di riferimento.

Dovendo prestar attenzione a tali aspetti si è pensato di far uso di un BPF_PERF_OUTPUT, un particolare tipo di mappa in cui è possibile inserire, dal programma BPF, degli "eventi" personalizzabili. Lato user space, restando in polling, è possibile intercettare e processare tali eventi richiamando una funzione handler.

5.5.2.1 Strutture dati

```
1. struct flow_pkt
2. {
3.     u16 filter_id;
4.     u16 pkt_id;
5.     u8 length;
6.     u16 netif_index;
7. };
8.
9. struct pkt_info
10. {
11.     u64 tstamp;
12.     u64 hash_low;
13.     u64 hash_high;
14. };
15.
16. struct stretch_event
17. {
18.     u16 filter_id;
19.     u16 netif_index;
20.     u8 mark;
21.     u8 length;
22.     u64 and;
23. };
24.
25.
26. BPF_PERF_OUTPUT(stretch);
27. BPF_HASH(flow_pkt_color0, struct flow_pkt, struct pkt_info);
28. BPF_HASH(flow_pkt_color1, struct flow_pkt, struct pkt_info);
```

Evento per la gestione dell'allungamento e strutture dati per le misure puntuali

5.5.2.2 Algoritmo

Nel nostro specifico caso, lato kernel, viene creato un evento di tipo "stretch", contenente le informazioni relative al flusso, e al valore di riferimento, che necessita l'intervento del programma in user space. In user space, si accede quindi direttamente ai valori interessati effettuando il controllo di validità.

Questo approccio segue ancora fedelmente quanto era stato ideato inizialmente e permette al programma di tracing nel kernel di continuare a processare i nuovi pacchetti, senza doversi preoccupare dei vecchi valori. Per queste motivazioni questa seconda

versione è sicuramente migliore della precedente. Vi è però un problema legato al cambio di contesto: quante più volte vengono generati eventi di tipo "stretch", tante più volte il sistema dovrà spendere del tempo nel cancellare i valori inutili. Il numero di cambi di contesto dipende però anche dal numero di flussi Il numero totale dei context switch può quindi essere non irrilevante e influire sulle prestazioni in maniera negativa.

5.5.3 Versione "loop-free"

5.5.3.1 Algoritmo

Partendo dai risultati dei programmi precedenti e analizzandone le problematiche, si è giunti alla conclusione di eliminare totalmente le operazioni di pulizia e controllo dei valori inutili, demandandola alla fase finale di lettura delle misure del blocco. Il programma eBPF continua quindi a inserire nuovi entries nelle mappe e ad incrementare l'*hash length*. Lato user space, a fine blocco, si leggeranno tutti i valori inseriti e si determineranno quali saranno validi con l'ultimo valore di riferimento usato, gli altri saranno invece scartati. Questa versione evita sia i problemi dei cicli legati lato kernel sia i cali di performance legati ai cambi di contesto. Le strutture dati sono le medesime della versione precedente.

Tale implementazione si allontana però dall'idea originale poiché il numero di campioni finali raccolti non è perfettamente predicibile. La prima volta vengono salvati un numero di campioni pari al valore massimo Max, dopo, per ogni nuovo incremento dell'*hash length,* Max/2. Questo scelta è basata sull'ipotesi di avere una elisione perfetta del 50% dei vecchi valori. Tale supposizione è ovviamente meramente teorica e determina quindi un numero di informazioni totali variabile al di fuori del range [Max/2, Max].

Attualmente è però, probabilmente, la soluzione più adatta sfruttando le caratteristiche di eBPF. Questo approccio alla pulizia "lazy" implica ovviamente un maggior consumo di memoria, ma le dimensioni del singolo inserimento sono di circa 50B, quindi anche a

fronte di milioni di inserimenti si può arrivare a qualche centinaio di MB. Tale memoria viene comunque periodicamente rimossa.

5.6 BPF Manager

Il BPF Manager, come già accennato nella descrizione architetturale, ha il ruolo di gestore dei programmi eBPF e permette di eseguire diverse operazioni:

- **Configuration**: ricevuto un oggetto, contente i parametri di configurazione, si occupa di preparare l'ambiente di esecuzione dei programmi di tracing, interagendo sia con le mappe, dove verranno salvati i filtri che identificano i flussi, sia con i programmi scritti in "C-like". Una volta completato il processo di inizializzazione, usando il BCC framework, verrà creato il bytecode del programma C-like e passato al "user space BPF verifier", un componente con il compito di controllare che non vi siano errori. Il programma non viene però ancora calato nella virtual machine né agganciato all' hook point.
- **Attach**: in un secondo momento, quando si riceve un messaggio indicante la volontà di avviare il programma di tracing, il programma eBPF viene effettivamente calato lato kernel e agganciato alla kprobe, ed è così pronto a esaminare i pacchetti in arrivo. E' da sottolineare che il programma eBPF può venire richiamato da più core, che stanno processando i pacchetti. Questo comportamento è intrinseco del funzionamento di eBPF, pertanto non è necessaria alcuna modifica al codice per godere dei vantaggi della scalabilità.
- **Detach**: il programma BPF viene rimosso, non verranno quindi raccolti ulteriori dati.
- **Read**: generica operazione di lettura delle mappe. Poiché esistono misure medie e puntuali si hanno due funzioni di lettura diverse. I dati vengono letti dalle mappe e salvati in apposite strutture dati a dizionario lato user space.

Si evidenzia nuovamente che le operazioni svolte dal BPF Manager non hanno al loro interno alcuna logica, si limitano ad effettuare le operazioni descritte. L'algoritmo che determina quali e quando usare tali funzioni è situato nel PNPM Manager.

5.7 PNPM Manager

Il PNPM Manager, oltre ad essere il punto di contatto tra l'interfaccia REST e il BPF Manager, implementa al suo interno la logica posta dietro alla metodologia PNPM. Le sue principali funzionalità sono:

- Creazione dell'oggetto BPF_Manager e passaggio della configurazione.
- Avvio della sessione di test pnpm. Questa fase comprende sia "l'attach" del programa eBPF sia la creazione di un thread parallelo per la lettura dei dati. Usando i parametri MP e MPC, indicanti rispettivamente la durata del singolo blocco e la durata totale del test, è possibile eseguire letture periodiche delle mappe, ove sono riportate le misure, ormai stabili, dei blocchi passati. I dati letti vengono poi salvati in strutture dati a dizionario, così che possano essere esportati. Concluse le operazioni di lettura, i dati nelle mappe vengono rimossi, cosicché quando arriverà un nuovo blocco le misure siano conteggiate correttamente.
- Esportazione delle misure lette.
- Pulizia delle proprie strutture dati e del BPF_Manager. Questa fase è necessaria per permettere l'inizio di una nuova sessione di test.

5.8 Configurazione e avvio programmi eBPF

Si evidenziano qui alcuni dettagli dei file relativi alla configurazione dei programmi di tracing e al loro avvio.

Tutti le tipologie di monitoraggio necessitano di alcune informazioni per poter essere usati secondo la logica PNPM.

```
1.  {
2.      "prog_id": 1,
3.      "mp":30,
4.      "mpc":120,
5.      "starter_mark":8,
6.      "next_mark": 16,
7.      "filters":
8.      [
```

```
{"id": 0, "dst_ip": "163.162.95.238"},
{"id": 1, "src_ip": "1.2.3.4", "dst_ip": "5.6.7.8", "proto": "udp", "src_port": 9, "dst_port": 9}
  ]
}
```

- **Prog_id:** seleziona la versione del programma di tracing da configurare.
- **Mp:** durata temporale del singolo blocco.
- **Mpc:** durata temporale dell'intero test.
- **Starter_mark:** valore del DSCP del primo blocco.
- **Next_mark:** valore del DSCP del secondo blocco.
- **Filters:** lista dei filtri, in cui è possibile usare tutti o un sottoinsieme dei campi disponibili.

Per la seconda e terza metodologia sono necessaria delle informazioni aggiuntive

```
{
    ...
  "match_value":249,
  "hash_length":2,
  "hash_function": "bob",
}
```

- **Match_value:** valore usato per la selezione dei pacchetti.
- **Hash_length:** indica quanti bit usare nel confronto tra hash e match_value.
- **Hash_function:** quale funzione di hash da usare.

Infine, solo per il terzo programma, è necessario indicare anche il valore massimo dei pacchetti entro cui mantenere hash length costante.

```
{
    ...
  "match_value":383146267,
  "hash_length": 0,
  "hash_function": "bob",
  "npkts": 8
}
```

E' possibile, come parametro opzionale, indicare quali interfacce monitorare.

```
{
...
  "netifs":
   [
       {"netif": "eth0", "netif": "eth1"}
   ]
}
```

5.9 RESTful API

Vengono qui brevemente spiegati i metodi esposti dall'interfaccia REST con i relativi parametri.

- **POST pnpm/**: viene inviata la configurazione per preparare il programma eBPF.
- **DELETE pnpm/<id>**: cancella il programma eBPF preparato in precedenza.
- **PUT pnpm/<id>/state**: avvia/arresta il programma eBPF.
- **GET pnpm/<id>/measures**: vengono restituite tutte le misure raccolte.
- **GET pnpm/<id>/measures/netif/<netif_id>/**: vengono restituite le misure raccolte sulla netif indicata.
- **GET pnpm/<id>/measures/flow/<flow_id>/**: vengono restituite le misure associate ad un certo flusso.
- **GET pnpm/<id>/measures/netif/<netif_id>/flow/<flow_id/**: vengono restituite le misure raccolte sulla netif e associate ad un certo flusso.

6 Validazione

Conclusa la fase implementativa si è passati alla valutazione della bontà dei programmi di tracing. Le soluzioni che sono state presentate sono dei prototipi, atti ovviamente a funzionare in contesti reali, ma vengono lasciate ad un futuro lavoro eventuali ottimizzazioni del codice.

Sono stati effettuati dei test di carico, al fine di misurare il numero massimo di pacchetti gestibili al secondo, e il relativo throughput, per ogni programma realizzato. Si è anche fatto uso di un programma di tracing "dummy", vuoto, come termine di paragone per il costo di processamento dei pacchetti. Per brevità si adotterà la seguente nomenclatura:

- **Programma 0**, programma vuoto. Non viene svolta alcuna operazione al suo interno.
- **Programma 1**, vengono usati tutti i pacchetti per aggiornare le misure.
- **Programma 2**, vengono usati tutti pacchetti il cui hash è uguale al valore di riferimento per l'aggiornamento delle misure legate al timestamp.
- **Programma 3**, si salvano i dati puntuali, in un numero compreso tra [~Max/2, Max ~], relativi ai pacchetti il cui hash è uguale al valore di riferimento.

Per i programmi due e tre è stata testata la funzione di hash BOB, in quanto è quella più adatta a processare pacchetti a velocità di linea.

Poiché eBPF, in modo nativo, permette di sfruttare a pieno CPU multi core, garantendo un'alta scalabilità, i test sono stati suddivisi in due categorie:

- **Test single core**, ove viene generato un flusso avente, per il layer 3 e 4, le stesse caratteristiche. Questo, in abbinamento alla tecnologia Receive-Side Scaling[1] (RSS)

[1] RSS è una tecnica hardware che permette la distribuzione dei pacchetti ricevuti su più core. La suddivisione viene effettuata in base al risultato di una funzione di hash, il cui input sono i campi del pacchetto.

presente sulla NIC usata, garantisce che un solo core venga usato per l'elaborazione.

- **Test multi core**, usando più flussi con caratteristiche diverse. RSS suddivide il traffico in arrivo su più code, associate ai diversi core. Ogni core innesca l'avvio del programma eBPF, permettendone un'esecuzione concorrente. L'accesso corretto alle mappe è garantito dai meccanismi Read-Copy-Update[1] (RCU) nativi.

Tutti i test sono stati effettuati presso i Telecom Italia Lab di Torino, installando IOVisor-PNPM su un server Huawei RH2288H V3, avente come processore un Intel Xeon E5-2690V3 @2.60GHz, 378GB RAM, scheda di rete 10 Gigabit Ethernet Intel® 82599ES, e sistema operativo Ubuntu 16.04 kernel 4.9. Per la generazione del traffico fino a 10Gbps è stato usato l'Agilent n2x collegato, mediante fibra ottica, al server.

6.1 Performance

6.1.1 Test single core

Agilent n2x è stato impostato per generare un singolo flusso di livello 3 per poter sfruttare un solo core. I filtri, nei programmi di tracing, sono stati scritti in modo congruente con i parametri scelti nella fase di configurazione del generatore di traffico.

[1] Meccanismo di sincronizzazione alternativo ai classici lock di lettura/scrittura.

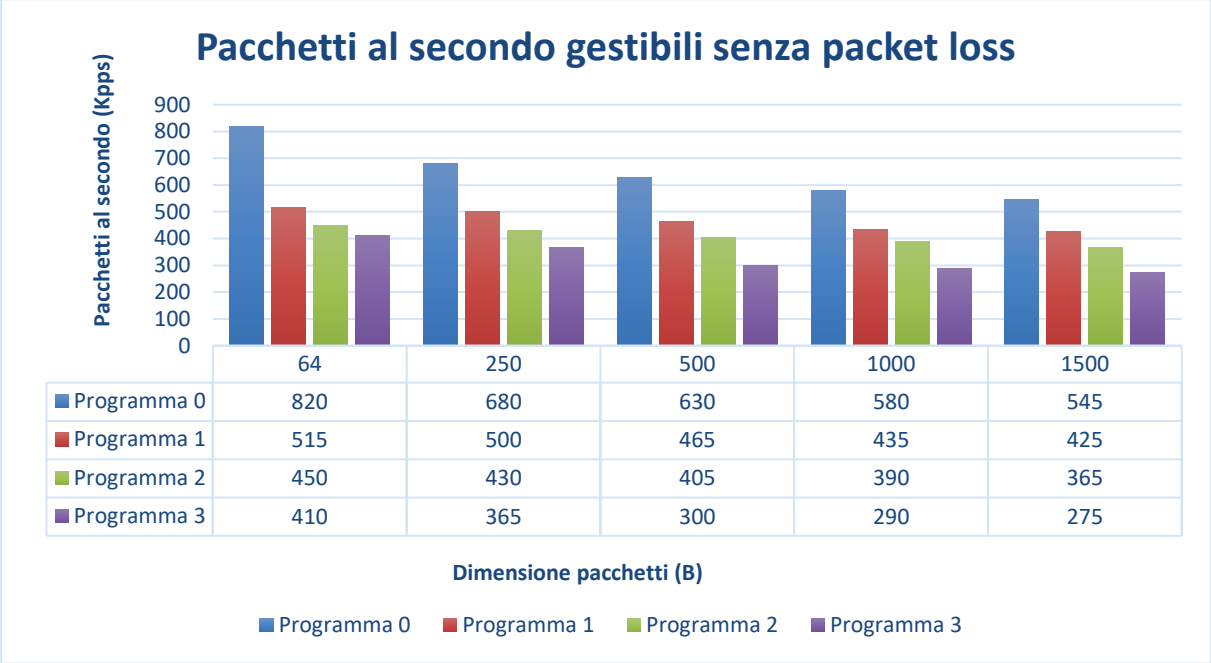


Figura 14 Pacchetti al secondo gestibili senza packet loss (single core).

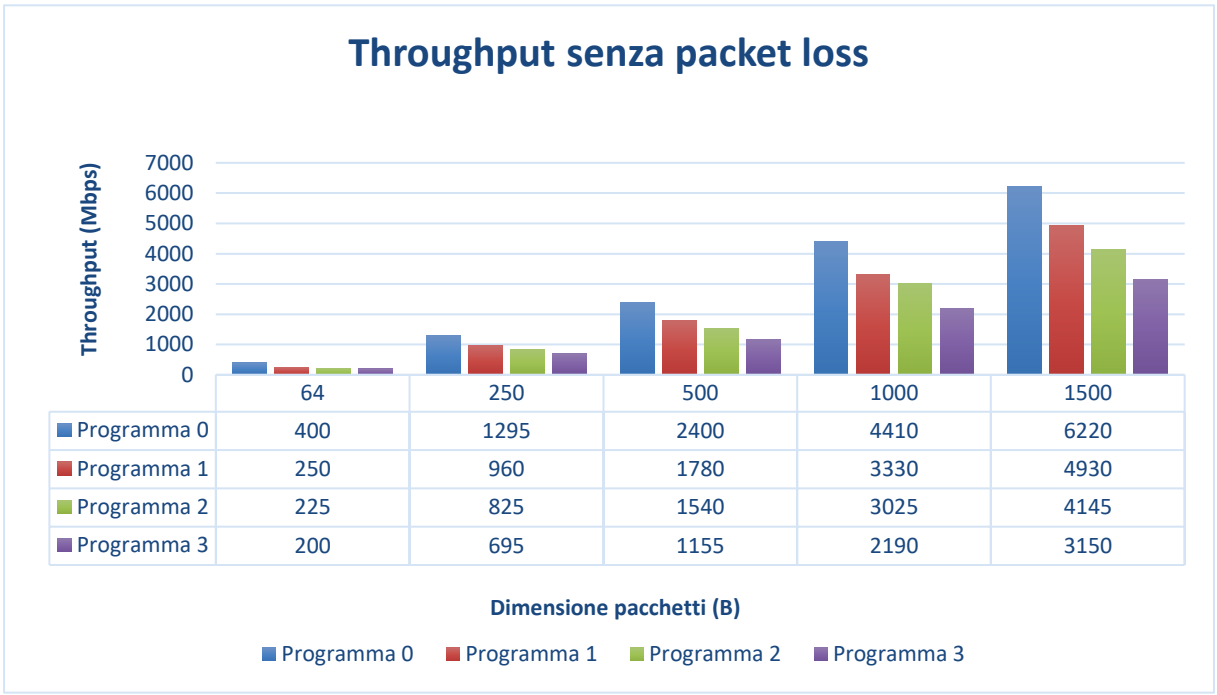


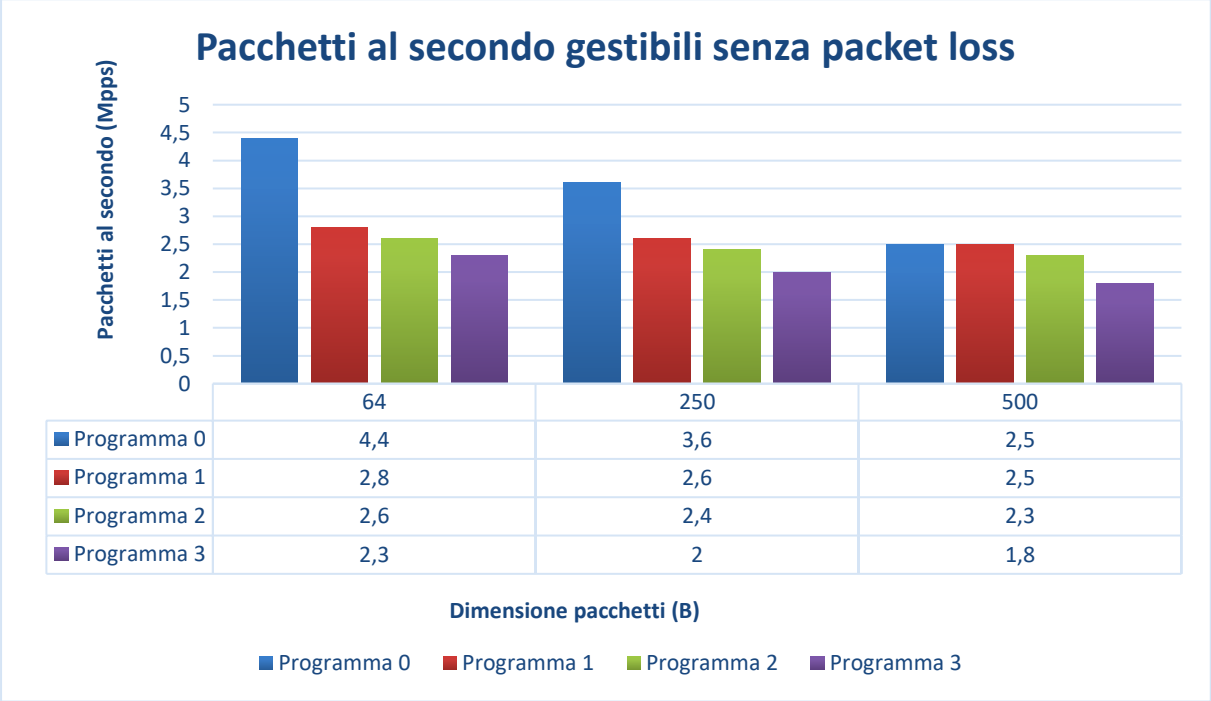
Figura 15 Throughput senza packet loss (single core).

I test sono stati condotti al variare delle dimensioni dei pacchetti, partendo da 64 fino a 1500B. I valori ottenuti con IOVisor-PNPM sono stati poi confrontati con quelli del generatore, al fine di verificare eventuali incongruenze.

Come si può notare, vi è un andamento decrescente all'aumentare della dimensione dei pacchetti, comportamento riconducibile alla maggiore velocità con cui vengono riempiti i buffer dedicati alla ricezione. Il grafico mostra anche le differenze che intercorrono fra le varie tipologie di programmi di tracing, mostrando come una maggior complessità nel codice implichi un minor tempo per il processamento delle informazioni ricevute, e quindi un minor numero di pacchetti gestibili.

6.1.2 Test multi core

Sono stati generati diversi flussi di livello 3 distinti in base all'indirizzo IP sorgente e destinazione. Grazie a RSS è stato possibile coinvolgere più core della CPU, permettendo quindi di suddividere l'elaborazione legata alla ricezione dei pacchetti.



	64	250	500
Programma 0	4,4	3,6	2,5
Programma 1	2,8	2,6	2,5
Programma 2	2,6	2,4	2,3
Programma 3	2,3	2	1,8

Figura 16 Pacchetti al secondo gestibili senza packet loss (multi core).

Risulta subito evidente l'aumento, fino a cinque volte, del numero dei pacchetti processabili. I test si fermano ai 500B in quanto, aumentando ulteriormente la dimensione dei pacchetti, il generatore utilizzato non era sufficiente per valutare con precisione i limiti dei programmi.

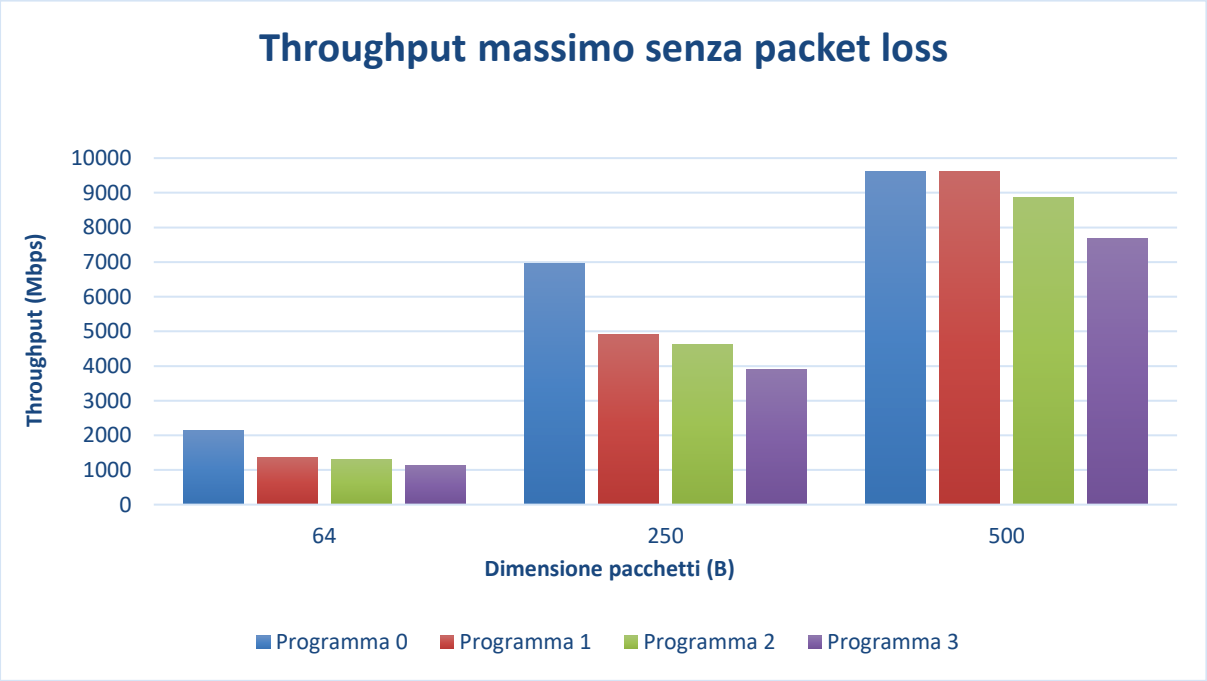


Figura 17 Throughput senza packet loss (multi core).

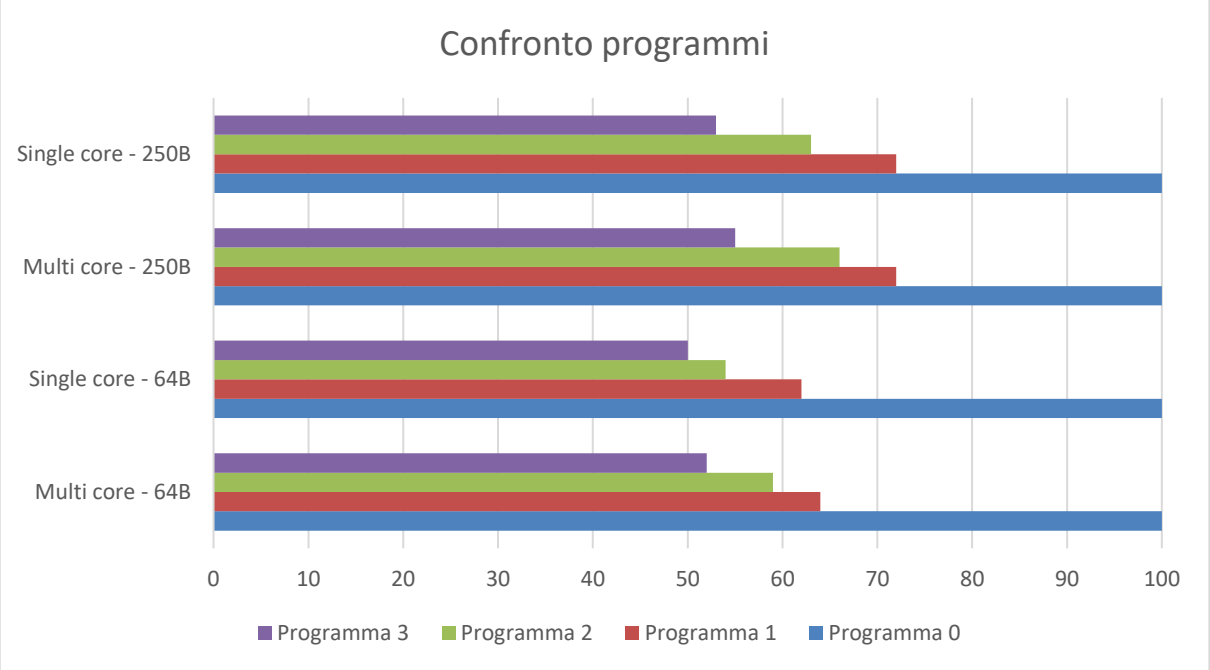


Figura 18 Confronto programmi single e multi core.

E' possibile inoltre osservare, usando come valore di riferimento il programma zero, che, nonostante il passaggio da single a multi core, le differenze tra le tipologie di programmi di tracing permangono. Questo conferma la nostra ipotesi che correlava le prestazioni alla complessità delle operazioni eseguite.

6.2 Analisi risultati

6.2.1 Packet loss

Individuati i valori limite per i programmi si è cercato di scoprire le cause legate al packet loss. Si è quindi fatto uso di Tcpdump, tool che permette la cattura e l'analisi dei pacchetti. La scelta è ricaduta su tale strumento in quanto il suo core è basato su BPF.

```
1. Packet captured: 890898
2. Packet received by filter: 6775964
3. Packet dropped by kernel: 5885066
4. Packet dropped by interface: 5426563
```

Esempio output tcpdump che evidenzia delle perdite di pacchetti.

Osservando l'output generato è possibile notare come un numero consistente di pacchetti è stato scartato dalla NIC. Questo tipicamente avviene quando il sistema non è "pronto" a processare i nuovi pacchetti in arrivo, a causa dei:

- Buffer pieni, in quanto i pacchetti sono smaltiti dal sistema troppo lentamente.
- Mancanze di risorse.

Parallelamente ai test di carico si è quindi osservato l'uso della CPU e si è scoperto che questa è la causa principale dei packet drop. Infatti quando uno o più core raggiungono il 100% d'uso è possibile che qualche pacchetto non venga conteggiato. La distribuzione dell'elaborazione permette di alleggerire il carico di lavoro sui singoli core, incrementando quindi la velocità di smaltimento dei pacchetti.

6.2.2 Confronto tipologie programmi

I risultati riportati sembrano contraddire la bontà delle tecniche di sampling applicate a PNPM, che dovrebbero ridurre il carico di lavoro andando a selezionare solo un sottoinsieme dei pacchetti. In realtà è necessario esaminare più nel dettaglio le differenze fra i tre programmi, concentrandoci in particolare sul numero di volte in cui il programma di tracing viene invocato e sul numero di accessi in memoria effettuati, operazione da sempre critica a livello prestazionale.

- **Il programma uno** si limita a riconoscere i flussi ed aggiornare le misure ad esse legate. Il numero delle sue esecuzioni e il numero di accessi in memoria è quindi legato al numero totale di pacchetti ricevuti.
- **Il programma due** effettua le operazioni di campionamento dei pacchetti ma deve sempre tenere aggiornati i contatori sul numero totale di pacchetti osservati. Questo significa che il numero di operazioni è pari a quello del programma uno. La differenza di performance è quindi dettata dalla funzione di hash utilizzata.
- **Il programma tre**, oltre a dover conteggiare, salva anche le informazioni legate ai singoli pacchetti, effettuando ulteriori N accessi in memoria, dove N è il numero di pacchetti selezionati. N dipende però da diversi fattori quali: numero di flussi identificati, numero di pacchetti il cui hash è pari al valore di riferimento, numero di "allungamenti" effettuati. Le performance sono quindi dipendenti dalla configurazione iniziale ed è quindi difficile stabilire un valore limite.

Dalla precedente analisi risulta evidente che il campionamento dei pacchetti in modalità tracing non offre vantaggi per ridurre il numero di pacchetti processati al secondo o il numero di accessi in RAM. Tali tecniche potrebbero invece offrire un vantaggio nell'acquisizione dei timestamp, altra funzionalità costosa per il sistema. Come è già stato detto in precedenza, con le attuali implementazioni, non è possibile acquisire un singolo timestamp, ma è necessario farlo generare dal sistema per tutti i pacchetti ricevuti.

7 Conclusioni e possibili sviluppi futuri

Mediante IOVisor è stato possibile creare un'implementazione di PNPM che superasse i limiti esposti a inizio trattazione, ovvero:

- Migliorare le performance rispetto ad un programma user space, in particolare in termini di pacchetti al secondo gestibili senza packet loss.
- Spostare il processamento dei pacchetti lato kernel per la gestione dei timestamp, usando la virtual machine integrata nei sistemi Linux.

Come è stato inoltre evidenziato nei test di carico multi core, che meglio simulano il traffico reale, i programmi realizzati riescono ad elaborare alcuni milioni di pacchetti al secondo, con un throughput superiore o prossimo ai 10Gbps. I valori ottenuti sono un buon primo risultato, in particolare per pacchetti di dimensione compresa tra i 500 e 1000B, che possono rappresentare uno scenario medio d'uso.

eBPF si mostra essere quindi un componente estremamente potente e versatile, dando la possibilità di iniettare codice runtime sicuro, grazie al BPF Verifier, garantendo al tempo stesso buone performance. IOVisor, grazie a BCC, semplifica lo sviluppo dei programmi, permettendo al programmatore di concentrarsi sulle funzionalità piuttosto che sul processo di creazione, compilazione e iniezione del codice. eBPF è ancora però una tecnologia sperimentale, in quanto sono presenti alcuni limiti nel nostro caso ben evidenziati dal problema nell'acquisizione di un timestamp. Per risolvere queste problematiche è necessario o rimediare a queste mancanze, usando quanto attualmente vi è a disposizione, o attendere l'uscita di nuove funzioni helper nelle versioni più recenti dei kernel Linux.

Attualmente i programmi due e tre, che usano le tecniche di hash sampling, non riescono quindi ad evidenziare i vantaggi ipotizzati all'inizio del lavoro. Eventuali sviluppi futuri, con l'aggiunta di un metodo per recuperare il valore temporale, potrebbero portare ad un aumento del numero di pacchetti processabili, senza la necessità di forzare il sistema a generare timestamp per tutti le informazioni ricevute. Tale novità implicherebbe

modifiche pressoché nulle nel codice, in quanto il valore sarebbe recuperato dalla funzione helper, e non più dalle strutture dati del kernel. Altro campo di indagine potrebbe essere l'uso dell'hardware timestamping, dove la generazione dei timestamp sarebbe demandata alla NIC. In questo scenario sarebbero da valutare oltre alle performance anche l'interazione con i programmi eBPF realizzati.

Sono poi ovviamente possibili miglioramenti nell'ottimizzazione del codice, specialmente nell'implementazione dei filtri, cercando di garantire maggiori performance mantenendo allo stesso tempo la possibilità dell'uso delle wildcard. La flessibilità offerta viene infatti controbilanciata dal numero di accessi in memoria necessari per associare un pacchetto ad uno specifico filtro.

In conclusione, IOVisor-PNPM rappresenta un buon punto di partenza per il monitoraggio delle prestazioni di rete secondo la metodologia PNPM. Il primo programma rimane attualmente quello più indicato a livello di performance, ma in futuro potrebbe essere possibile sfruttare appieno i vantaggi offerti dal campionamento dei pacchetti, incrementando ulteriormente i pps processabili e richiedendo meno risorse all'apparato.

Bibliografia

[1] A. Capello, M. Cociglio, G. Fioccola, L. Castaldelli, *A packet based method for passive performance monitoring.* URL: https://tools.ietf.org/html/draft-tempia-ippm-p3m-03.

[2] M. Chen, L. Zheng, G. Mirsky, G. Fioccola, T. Mizrahi, *IP Flow Performance Measurement Framework.* URL: https://tools.ietf.org/html/draft-chen-ippm-coloring-based-ipfpm-framework-06.

[3] S. Bryant, G. Swallow, S. Sivabalan, *Synonymous Flow Labels.* URL: https://tools.ietf.org/html/draft-bryant-mpls-synonymous-flow-labels-00.

[4] S. McCanne, V. Jacobson, *The BSD Packet Filter: A New Architecture for User-level Packet Capture.* In: USENIX winter. Vol. 46. 1993.

[5] Tcpdump e LibPcap. URL: http://www.tcpdump.org/.

[6] WinPcap. URL: https://www.winpcap.org/.

[7] Wireshark URL: https://www.wireshark.org/

[8] J. Corbet, *Extending extended BPF.* URL: https://lwn.net/Articles/603983/.

[9] *IO Visor Project.* URL: https://www.iovisor.org/.

[10] T. Zseby, M. Molina, N.G. Duffield, S. Niccolini, F. Raspall, *Sampling and Filtering Techniques for IP Packet Selection.* URL: https://tools.ietf.org/html/rfc5475.

[11] M. Molina, S. Niccolini, N.G. Duffield, *A Comparative Experimental Study of Hash Functions Applied to Packet Sampling.,* ITC-19, Beijing, 2005. URL: https://pdfs.semanticscholar.org/896d/42232d04e7ab04cbdca80fe56b436a5ff381.pdf

[12] M. Grossglauser, J. Rexford, *Passive traffic measurements for IP operations.,* Oxford University Press, 2003. URL: http://www.cs.princeton.edu/~jrex/knowplane/.

[13] N.G. Duffield, S. Niccolini, M. Molina, Quittek J., *Uniformity and Independence of Hash Functions for Packet Sampling.*, IETF 64, Novembre 2005. URL: https://www.ietf.org/proceedings/64/slides/psamp-1.pdf.

[14] P. Orosz, T. Skopko, *Performance Evaluation of a High Precision Software-based Timestamping Solution for Network Monitoring.*, International Journal on Advances in Software, Vol. 4, 2011, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.361.9968&rep=rep1&type=pdf#page=189.

[15] D. P. Bovet, M. Cesati, Understanding the Linux Kernel., 3 ed. O'Reilly, Novembre 2005, capitolo 6.

[16] V. Moreno, P.M. Santiago del Rio, J. Ramos, J.J.Garnica, J.L. Garciıa-Dorado, *Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines.,* IEEE, Ottobre 2012.

[17] V. Hsiao, *Meet cute-between eBPF and Kernel Tracing.,* URL: https://www.slideshare.net/vh21/meet-cutebetweenebpfandtracing.

[18] F. Sanchez, *Extended BPF and Data Plane Extensibility: An overview of networking and Linux.,* Red Hat Summit, Giugno 2015. URL: https://videos.cdn.redhat.com/summit2015/presentations/13737_an-overview-of-linux-networking-subsystem-extended-bpf.pdf.

[19] Tipologia mappe eBPF URL: http://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html

[20] T. Graf, *Next Generation of Programmable Datapath.*, OVS Conference, 2016. URL: http://openvswitch.org/support/ovscon2016/7/1030-graf.pdf.

[21] *BPF Compiler Collection*. URL: https://github.com/iovisor/bcc.

[22] *Funzionamento eBPF con kprobe* URL: https://github.com/iovisor/bpf-docs/blob/master/bpf-internals-2.md.

[23] E. Nahum, *Buffer management*, COMS W6998, Primavera 2010.