UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Utilizzo di eBPF per l'analisi del traffico di rete di sistemi Linux

Candidato:
Riccardo Colavita

Relatore:
Luca Deri

Anno Accademico 2016-2017

Sommario

Per uno sviluppatore o un manager di rete, nessuno strumento è così prezioso come lo è un monitor di rete. Questo strumento permette di catturare, visualizzare e tenere traccia dei pacchetti destinati ai propri host; tracciare i pacchetti aiuta a capire più facilmente per quale motivo due host non riescono a comunicare, o perché le performance della rete sono basse.
La maggior parte dei monitor di rete disponibili in commercio sono unità indipendenti dedicate al monitoraggio di protocolli specifici.

Lo scopo di questo elaborato è quello di presentare un nuovo strumento utilizzato per l'analisi del traffico di rete di sistemi Linux, che permette un filtraggio di pacchetti con un alto grado di flessibilità, buone performance e sopratutto basso overhead. A questo proposito sono stati sviluppati nuovi tool per l'analisi del traffico di rete e per il monitoraggio di sistema in ambiente Linux.

Una volta introdotte le varie tecniche e gli strumenti necessari per lo sviluppo di tool di monitoraggio presenteremo in linea generale la struttura e il workflow di mini-programmi da me sviluppati per monitorare le reti. In particolare ci concentreremo su di uno strumento chiamato eBPF, che verrà utilizzato per lo sviluppo di due programmi utili per testare la latenza di rete e per vedere quali sono le applicazioni web che causano errori di paginazione in memoria.

Indice

Capitolo 1

Introduzione

Unix è diventato il sistema di riferimento per quel che riguarda il monitoraggio della rete, oggigiorno gli utenti Unix richiedono affidabilità e reattività per accedere ai servizi della rete. Per garantire questi tipi di performance bisogna utilizzare dei meccanismi efficienti per il demultiplexing dei pacchetti ricevuti [13].
Quando un pacchetto viene ricevuto da un nodo della rete, che si trova a metà o alla fine di collegamento, occorre determinare rapidamente e in modo efficiente il/i destinatario/i dei dati, accettando, inoltrando oppure scartando i pacchetti ricevuti. Tali determinazioni possono essere fatte da un controller hardware, da un controller software o da una combinazione di entrambi.
Nelle reti di tipo broadcast, durante l'analisi del traffico, ogni nodo è responsabile dell'accettazione dei pacchetti di "interesse", respingendo tutti gli altri. Questo meccanismo viene chiamato "filtraggio di pacchetti"[3]. Negli ultimi decenni, a seguito di alcune ricerche innovative [2] si è arrivati a ridefinire il concetto di filtraggio di pacchetti, che nella sua forma più semplice non è altro che un'astrazione di una funzione, o predicato booleano, applicata ad un determinato flusso di pacchetti, permettendo di selezionare uno o più specifici sottoinsiemi di pacchetti appartenenti a tale flusso.

Lo scopo di questo elaborato è quello di presentare un nuovo strumento utilizzato per l'analisi del traffico di rete che ci consente, grazie al supporto di alcuni moduli presenti nel kernel di Linux, di catturare i pacchetti di interesse utilizzando un concetto chiamato *filtraggio degli eventi*.

1.1 Trace filtering

Con il tradizionale approccio di debug, diventa molto difficile recuperare dettagli accurati sul comportamento di sistemi real-time o soft real-time, per tanto viene impiegato un meccanismo di registrazione veloce di dati chiamato *tracing*. In [15] si fà distinzione tra tracing statico e tracing dinamico in base al loro aspetto funzionale o in base al loro uso (kernel o userspace tracing).
Il tracing comporta l'aggiunta di *tracepoint* nel codice. Un tracepoint non è altro che una funzione inserita nel codice (in caso di applicazioni utente), oppure può far parte dell'infrastruttura del kernel (tracepoint "hook" nel kernel di sistemi Linux). Ogni tracepoint è associato ad un evento. Ad ogni istante gli eventi nel kernel di sistemi Linux occorrono frequentemente e sono di basso livello[1].
Per le applicazioni utenti i tracepoint possono essere qualsiasi funzione chiamata da un loro stesso programma. Questo ci permette di poter monitorare un programma in esecuzione in maniera molto efficiente, piuttosto che con il debugging tradizionale in scenari in cui l'effetto di mettere in pausa, o di attendere l'interazione dell'utente e raccogliere i dati, possono alterare il comportamento di un'esecuzione normale e produrre risultati di analisi errati. In particolare, il meccanismo di tracing del kernel[5] consente ad un utente opportunamente privilegiato di ricevere informazioni dettagliate ogni volta che l'esecuzione di un suo programma passa attraverso il tracepoint del kernel. Come si può immaginare, la quantità di dati che vengono ricevuti su ogni tracepoint può essere abbastanza grande. Bisogna quindi aggiungere dei meccanismi che consentono di catturare solamente i pacchetti di interesse.

1.2 Packet capture

Molte versioni di Unix forniscono servizi a livello utente per la cattura dei pacchetti, rendendo possibile l'uso di postazioni di lavoro generiche per il monitoraggio della rete.
Come specificato in [11], un modo efficiente per far si che i monitor di re-

[1] Alcuni esempi di eventi possono essere le syscall `fork()`/ `exec()` o scheduling calls, ecc..

te[2] riescano a copiare i pacchetti attraversando il limite kernel/user-space, è quello di minimizzare la copia dei pacchetti attraverso l'utilizzo di un kernel agent chiamato *packet filter*, che copia i pacchetti di interesse e scarta i pacchetti indesiderati il prima possibile.

Quando un pacchetto arriva ad un'interfaccia di rete, i driver che si trovano a livello collegamento dello stack TCP/IP, lo inviano al system-protocol stack. Ma se il kernel agent è in ascolto su quella interfaccia, il driver prima di inviarlo al system-protocol richiama l'attenzione del kernel agent[3]. In questo caso entra in gioco il filtro definito a livello utente che decide quali pacchetti il kernel agent deve copiare e di conseguenza quali scartare.
Poiché i monitor di rete spesso vogliono solo un sottoinsieme di informazioni relative al traffico di rete, si ottiene un notevole guadagno di prestazioni catturando i soli pacchetti di interesse. Per minimizzare l'utilizzo della memoria (il vero e proprio collo di bottiglia nella maggior parte dei moderni workstation), il pacchetto deve essere filtrato "sul posto", piuttosto che copiato in un buffer del kernel prima di essere filtrato. Così, se i pacchetti non sono accettati, solo i byte che erano necessari per il processo di filtraggio sono recuperabili dall'host.

[2]eseguiti come processi a livello utente.

[3]In questo momento è la componente chiamata *Network Tap* (NT) del kernel agent che colleziona le copie dei pacchetti che arrivano dalle interfacce di rete, ed è esso stesso che le invia verso le applicazioni in ascolto.

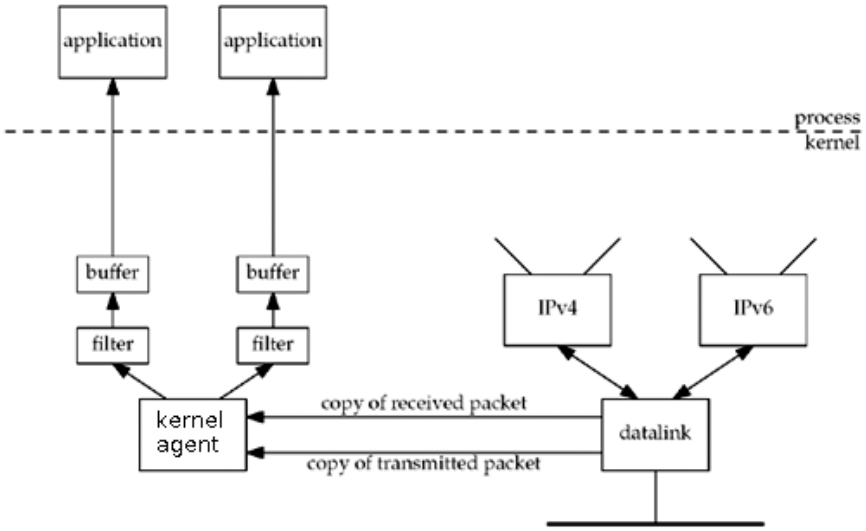


Figura 1.1: Mostra come il kernel agent è strettamente correlato alle altre parti del sistema. I pacchetti ricevuti dalla rete sono passati attraverso il packet filter e distribuiti alle applicazioni

1.3 Motivazione e obiettivo del lavoro

La maggior parte degli strumenti di monitoraggio si basano su meccanismi che offrono prestazioni diverse sotto vari aspetti e scenari. Nelle sezioni seguenti, saranno analizzati alcuni fattori che influenzano le prestazioni dei tool per il monitoraggio di rete, ed infine andremo a presentare l'obiettivo del lavoro svolto in questo elaborato.

1.3.1 In mancanza di visibilità

A differenza di come sono implementati i protocolli della rete, le applicazioni di monitoring, di solito, sono interessate solo a pochi pacchetti, o nello specifico solo ad alcuni byte in essi contenuti. Applicazioni che tengono traccia delle connessioni per trovare un bug nel protocollo di rete, o che provano a misurare il carico effettivo di una rete, devono evitare di perdere pacchetti, in modo da poter garantire delle misure affidabili e dettagliate sul traffico di rete. In questo caso i kernel agent devono ovviare a questo problema utilizzando delle code limitate che possono essere gestite dall'applicazione utente. Inoltre nella rete manca spesso visibilità di alcune parti come comunicazioni

intra-macchina, intra-VM e intra-container. Ad esempio in caso di comunicazione intra-container[4] bisogna utilizzare delle tecniche che permettono, all'applicazione di monitoraggio, di poter osservare dall'alto l'intera infrastruttura del sistema, consentendo quindi di diagnosticare anche eventuali problemi con l'app containerizzata.

1.3.2 Metriche più realistiche e dettagliate

Alcuni tipi di applicazioni possono richiedere delle statistiche affidabili, ad esempio sul tempo che passa dall'istante in cui il pacchetto lascia il canale fino all'istante in cui esso è stato ricevuto, e poiché il processo di schedulazione viene eseguito in base alla politica dello schedulatore, può passare del tempo tra la ricezione di un pacchetto e la sua elaborazione da parte dell'applicazione. Ciò significa che i pacchetti devono essere marcati temporalmente dal kernel, in un punto coerente durante il servizio di interrupt; in caso contrario, è impossibile assegnare in modo affidabile ad un pacchetto la tempistica in cui esso è stato ricevuto[12].
In questo modo se le misure ricavate dal processo di monitoraggio siano poco performanti, e se si conosce il processo[5], si può facilmente risalire al problema.

1.3.3 Obiettivo del lavoro

L'attuale stato dell'arte per l'analisi del traffico di rete non fornisce una soluzione ottimale. In termini di prestazioni, il fattore più importante in tutti gli strumenti di monitoraggio, oltre a quelli discussi nelle due sezioni precedenti, è quello di ridurre l'overhead legato all'elaborazione dei pacchetti [15].

Lo scopo di questo elaborato è quello di presentare, attraverso lo sviluppo di alcuni tool, uno strumento di monitoraggio per l'analisi del traffico di rete in sistemi Linux, che sia efficiente in termini di overhead e che sia in grado di fornire misure dettagliate e affidabili per l'analisi del traffico.

[4]Si veda progetto WeaveScope. https://www.weave.works/docs/scope/
[5]In questo caso ci riferiamo al PID o al nome simbolico del processo.

1.4 Struttura della tesi

La tesi è strutturata nel seguente modo:

- **Stato dell'arte:** Nel capitolo 2 viene presentato l'attuale stato dell'arte degli strumenti di monitoraggio delle reti.
- **Progetto:** Nel capitolo 3 viene descritto il funzionamento di eBPF da un punto di vista progettuale, partendo dall'analisi dei requisiti necessari per lo sviluppo di programmi eBPF, saranno poi presentati due tool di monitoraggio da me sviluppati.
- **Validazione e Risultati:** Nel capitolo 4 verranno validati i tool sviluppati, a partire dall'analisi dei requisiti fatta nel capitolo precedente, cercando di capire quali problemi risolvono; verranno poi presentati i risultati finali per ognuno dei due tool e alla fine del capitolo vedremo, in linee generali, il lavoro futuro legato all'espansione dei due strumenti.
- **Conclusioni:** Nel capitolo 5 presentate le conclusioni del lavoro svolto.

Capitolo 2

Stato dell'arte

Negli ultimi anni, un certo numero di sforzi di ricerca innovativi hanno costruito l'uno sull'altro un nuovo concetto di packet filtering. La prima proposta fù fatta da Mogul, Rashid e Accetta nel 1987 [13]; un packet filter era definito come una astrazione di un predicato booleano programmabile dall'utente. Mentre questo modello di filtro è stato pesantemente sfruttato per il monitoraggio della rete, la raccolta del traffico, la misurazione delle prestazioni e demultiplexing del protocollo a livello utente, più recentemente, è stato proposto il filtraggio per la classificazione dei pacchetti nei router (ad esempio per servizi real-time), filtri firewall [10] ed intrusion detection.
Le prime rappresentazioni di packet filter erano basate su un modello di esecuzione imperativo. In questa forma, un packet filter è rappresentato come una sequenza di istruzioni che sono conformi ad alcune macchine virtuali, allo stesso modo dei moderni Java byte code che possono essere eseguiti su una macchina virtuale Java.

L'allacciamento di un network device alla rete viene realizzato con il supporto di un "controller" che lavora indipendentemente dal processore host. Il filtraggio dei pacchetti avviene quindi in due fasi successive a partire dal controller, che esamina i pacchetti in tempo reale. Per fare ciò il controller è "condizionato" da un sottoinsieme appropriato di criteri specifici di filtraggio, in base alle capacità di filtraggio di quel controller[3].
Il controller classifica i pacchetti in tre categorie:

- quelli che non soddisfano i criteri di filtro ("da respingere");

- quelli che soddisfano i criteri ("corrispondenze esatte");
- quelli che soddisfano parzialmente i criteri ("corrispondenze parziali").

I pacchetti scartati non vengono consegnati ad un processo di elaborazione, relativo al monitoraggio. Mentre pacchetti che sono classificati come corrispondenze esatte o parziali vengono consegnate al processo di monitoraggio in modo da poterli analizzare.
Le implementazioni dei controller sono limitate dal fatto che devono elaborare i pacchetti in tempo reale appena essi vengono ricevuti. Queste limitazioni pongono un alto valore nella ricerca di meccanismi di filtraggio che possono essere implementati con un minimo di logica e memoria.

Osservazione.
Una prima considerazione è l'efficienza ($\mathcal{E}$) del filtro, che in questo contesto può essere espressa come:

$$\mathcal{E} = \frac{I_c}{P_c}$$

dove:
- I_c : È il numero di pacchetti di interesse;
- P_c : E il numero di pacchetti effettivamente processati.

Con $\mathcal{E} = 1,0$ si rappresenta l'efficienza di filtraggio esatta in cui ogni pacchetto candidato per il processo di elaborazione è un occorrenza di nostro interesse.

In questo capitolo vengono descritti i principali strumenti di monitoraggio di rete.

2.1 Sysdig

Sysdig [8] è uno strumento open-source per il monitoraggio di sistemi Linux. Cattura chiamate di sistema ed eventi Linux usando una funzione del kernel chiamata tracepoints. Richiede che un driver chiamato sysdig-probe sia stato caricato nello spazio del kernel per poter registrare i risultati dei tracepoints relativi a chiamate di sistema. Le informazioni vengono "impacchettate" e

mandate al processo in spazio utente, oppure vengono salvate nei trace file[1]. Le informazioni catturate nei trace file possono essere filtrate usando le estensioni che mette a disposizione sysdig. Questo permette report più specifici sullo stato corrente dei sistemi, ad esempio sulla CPU, sulle attività I/O e sull'utilizzo della rete di un determinato processo.
Attualmente, sysdig registra i risultati dei tracepoints delle chiamate di sistema in ingresso e in uscita, e registra gli eventi del processo di monitoraggio da noi pianificati. Il gestore, chiamato Sysdig-probe, per questi eventi è molto semplice: si limita alla copia dei dettagli dell'evento in un buffer condiviso, codificato per un consumo successivo. La ragione per cui bisogna mantenere il gestore semplice e minimale, come si può immaginare, è legata alle performance, dal momento che l'esecuzione del kernel è "congelata" fino a quando il gestore non termina.
Questo è tutto quello che fa il driver, il resto avviene tutto a livello utente. La figura 2.1 mostra l'architettura generale dello strumenti in esame.

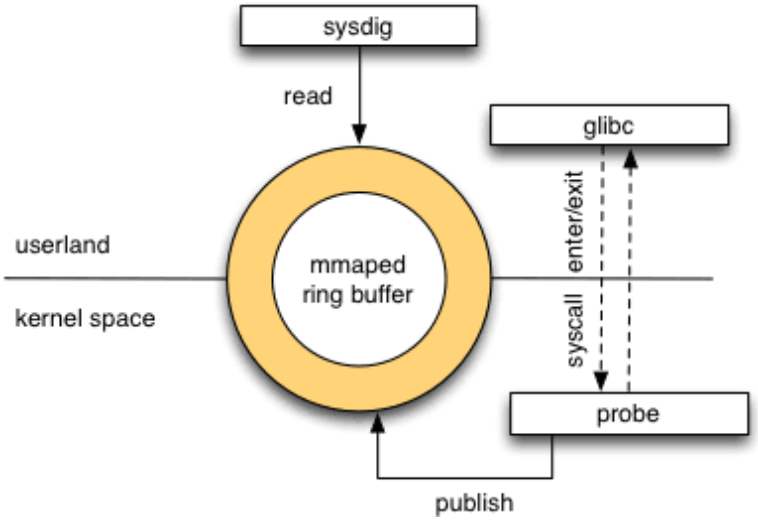


Figura 2.1: Mostra l'architettura del tool di monitoraggio Sysdig.

[1]Simile al modo in cui tcpdump analizza i pacchetti di rete

Il buffer degli eventi è mappato in memoria nello spazio utente in modo che sia possibile accedervi senza effettuare alcuna copia, riducendo al minimo l'utilizzo della CPU e gli errori di cache. Due librerie, `libscap` e `libsinsp`, offrono quindi supporto per la lettura, la decodifica e l'analisi degli eventi. In particolare, `libscap` gestisce i trace file, mentre `libsinsp` include funzionalità sofisticate di tracciamento dello stato (ad esempio è possibile utilizzare un nome di un file anziché il suo FD), filtraggio, decodifica eventi, un compilatore Lua JIT [1] (Just-In-Time) per eseguire i chisels[2] e molto altro. Infine, sysdig lo completa come un semplice wrapper attorno a queste librerie.

2.1.1 Considerazioni finali sullo strumento

Ma cosa succede se sysdig, `libsinsp` o `libscap` non sono abbastanza veloci per stare al passo con il flusso di eventi provenienti dal kernel? Sysdig rallenterà il sistema? In questo scenario, il buffer degli eventi si riempie e sysdig-probe inizia a rilasciare gli eventi in arrivo. Quindi verranno perse un pò di informazioni sul monitoraggio, ma la macchina e gli altri processi in esecuzione non verranno rallentati.
Questo è il vantaggio chiave dell'architettura di Sysdig. L'overhead di tracing ($\mathcal{E}$) in questo caso è molto prevedibile e significa che Sysdig è uno strumento ideale per il monitoraggio della rete.

2.2 Windows Minifilter driver Capture

In questa sezione viene presentato uno strumento di analisi su sistemi Windows.
Capture è uno strumento presentato in [14] che viene utilizzato per il monitoraggio di rete e di sistema per Windows.
Analogamente agli altri strumenti esistenti, Capture analizza lo stato del sistema operativo e delle applicazioni che vengono eseguite sul sistema controllando il file system, il registro di Windows, e il process monitor, in modo da poter generare report per tutti gli eventi ricevuti dai tre monitor. I tre monitor sono descritti sotto:[width=2cm

[2] I chisels sono piccoli script riutilizzabili scritti in Lua.

- **File system monitor**: acquisisce tutti gli eventi di lettura o scrittura nei file system montati mentre gli eventi si verificano sul sistema, incluse informazioni su quando si è verificato l'evento, il tipo di evento, il processo che ha attivato questo evento e il nome completo del file o directory su cui è stato applicato

- **Registry monitor**: acquisisce un insieme simile di eventi, ma si concentra sul registro di Windows, che memorizza le opzioni di configurazione del sistema operativo e delle applicazioni installate in una grande tabella hash. Il monitor del registro riporta l'ora con una risoluzione in millisecondi, il processo che ha attivato l'evento di registro, il path della chiave in cui si è verificata l'azione, e il tipo di azione eseguita sulla chiave. Il registro di Windows consente all'utente di leggere e manipolare le coppie chiave/valore e permette di scoprire il contenuto di un particolare percorso del registro.

- **Process monitor**: monitora la creazione e la distruzione di processi, ma non riporta nessuna informazione sui processi già in esecuzione. Con l'arrivo di ogni evento cattura le informazioni del processo creato o distrutto, salvandole in un file. Inoltre il Process monitor cattura le informazioni del processo genitore che aiutano a determinare quale processo ha creato o distrutto il processo che ha causato l'attivazione dell'evento.

Capture consiste nell'uso di due componenti, come lo si può vedere in figura 2.2, un insieme di kernel-driver e un processo in spazio utente.
I driver del kernel operano solo nello spazio del nucleo del sistema operativo, e utilizzano meccanismi di tracing basati sugli eventi per monitorare lo stato del sistema. Mentre il processo in spazio utente, che comunica con i driver del kernel, filtra gli eventi in base agli output.
Il client Capture utilizza i driver del kernel per monitorare il sistema usando il meccanismo di callback del kernel esistente che avvisa i driver registrati quando si verifica un determinato evento. Queste callback richiamano funzioni all'interno di un driver del kernel e passano le informazioni dell'evento in modo che possano essere modificate o, nel caso di Capture, monitorate.

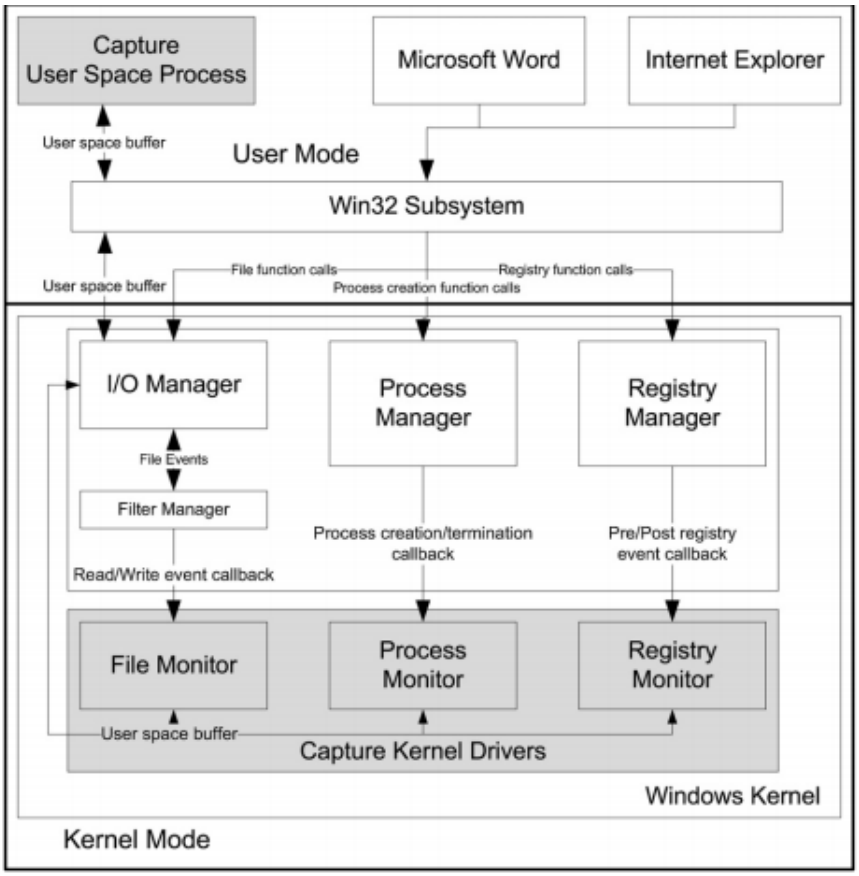


Figura 2.2: Architettura di Capture

Quando i driver del kernel ricevono gli eventi, li inseriscono in un coda in attesa di essere inviati, attraverso un buffer[3], al client Capture che si preoccuperà di elaborare e filtrare le informazione ricevute dai driver.

2.2.1 Considerazioni finali sullo strumento

Nella sezione precedente è stato presentato uno strumento open-source per il monitoraggio di sistemi Windows, presentando l'architettura e le funzionalità dello strumento che soddisfano le esigenze di un analista. Capture offre strumenti per la visualizzazione e rilevazione dei problemi di sistema o di rete. Inoltre permette di ricevere le statistiche degli eventi generati dai processi

[3]Il buffer viene allocato dal client Capture e passato ai driver che si trovano nel kernel. Il passaggio del buffer si verifica tramite l'API Win32 e il gestore I/O

in esecuzione, attraverso l'utilizzo di un buffer che mette in comunicazione il processo in spazio utente e il kernel del sistema operativo. Cosa succede se il buffer si riempie? In questo caso si posso perdere alcune statistiche sugli eventi generati. Quindi per ottenere risultati ottimali, in termini di efficienza $\mathcal{E}$, bisogna creare un buffer sufficientemente grande da poter contenere il traffico da analizzare, o memorizzare solo un parte dell'informazione contenuta nell'evento verificatosi. Risulta quindi essere uno strumento di monitoraggio molto meno efficiente descritto nella sezione 2.1.

2.3 eBPF (extended Berkeley Packet Filter)

L'originale Berkeley Packet Filter (BPF) [11] è stato progettato per catturare e filtrare i pacchetti di rete che corrispondono a regole specifiche. I filtri vengono implementati come programmi da eseguire su una macchina virtuale basata su registri.
La possibilità di eseguire programmi dello spazio utente all'interno del kernel si è rivelata una decisione di progettazione utile, ma altri aspetti del progetto BPF originale, cBPF (classic-BPF), non sono stati così validi. Per prima cosa, il progetto della macchina virtuale e la sua architettura, designata delle istruzioni ISA (Instruction Set Architecture), è stato lasciato indietro mentre i processori moderni si trasferivano nei registri a 64 bit e inventavano nuove istruzioni richieste per i sistemi multiprocessore, come l'istruzione di scambio atomico (XADD).
Inizialmente [11] BPF consisteva nell'iniettare un semplice bytecode dallo spazio utente nel kernel, dove veniva controllato da un verifier, per evitare crash del kernel o problemi di sicurezza, e mandato in esecuzione ogni volta che la socket a cui era collegato riceveva un pacchetto. L'attenzione di BPF nel fornire un numero limitato di istruzioni RISC non corrispondeva più alla realtà dei processori moderni [6].
Alexei Starovoitov ha introdotto il design di BPF, eBPF (extend-BPF), per sfruttare i progressi dell'hardware moderno. La semplicità del linguaggio e l'esistenza di macchine di compilazione JIT presenti nel kernel, sono stati per BPF, degli ottimi fattori per quel che riguarda le prestazioni di questo strumento.

Negli ultimi anni sono apparse nuove funzionalità come l'introduzione di mappe e tail calls. Le macchine di compilazione sono state riscritte e il nuovo linguaggio e ancor più vicino al linguaggio macchina nativo rispetto a cBPF. Inoltre sono stati creati nuovi tracepoint nel kernel.
La tracciabilità dei sistemi Linux è di conseguenza migliorata. Strumenti come *ftrace* e *perfevents* [7] fanno ora parte del kernel di Linux, e permettono di analizzare le prestazioni di Linux in dettaglio e di risolvere problemi legati al processore, al kernel e alle applicazioni.
Grazie a queste nuove funzionalità, i programmi eBPF possono essere designati per vari casi d'uso, che si dividono in due campi di applicazione. Uno di questi è il dominio del tracing kernel e del monitoraggio degli eventi. I programmi BPF possono essere associati a tracepoint e possono essere confrontati con altri metodi di tracing.
L'altro dominio di applicazione è riservato all'analisi del traffico di rete. Oltre ai filtri applicabili alle socket, i programmi eBPF possono essere allacciati direttamente alle interfacce di ingresso o uscita di tc (Linux Traffic Control tool), ed eseguiti su varie attività di elaborazione di pacchetti, in modo efficiente.

Nelle sezioni che seguono viene presentata l'architettura e le funzionalità di supporto di eBPF.

2.3.1 Architettura e set di istruzioni

Il classico BPF (cBPF) aveva due registri da 32 bit: A e X. Tutte le operazioni aritmetiche erano eseguite con il supporto di questi due registri.
Uno dei cambiamenti più importanti è stato il passaggio ai registri da 64 bit e un aumento del numero di registri da due a dieci. Poiché le architetture moderne hanno più di due registri, ciò consente di passare i parametri alle funzioni nei registri di macchine virtuali eBPF, proprio come su hardware nativo [6]. Il BPF più recente (eBPF) ha 10 registri da 64 bit e supporta load/store arbitrarie. Contiene anche nuove istruzioni come `BPF_CALL` che possono essere usate per chiamare alcune nuove funzioni del kernel chiamate helper. Analizzeremo questo dettaglio nelle prossime sezioni. Il nuovo eBPF

supporta chiamate convenzionali che sono più simili alle macchine moderne (x86_64). Ecco la mappatura dei nuovi registri eBPF nei registri x86:

```
R0   -  rax     return value from function
R1   -  rdi     1st argument
R2   -  rsi     2nd argument
R3   -  rdx     3rd argument
R4   -  rcx     4th argument
R5   -  r8      5th argument
R6   -  rbx     callee saved
R7   -  r13     callee saved
R8   -  r14     callee saved
R9   -  r15     callee saved
R10  -  rbp     frame pointer
```

Il registro `R0` memorizza il ritorno del programma eBPF, mentre gli argomenti del programma possono essere caricati attraverso `R1-R5`.
eBPF quindi non è altro che un set di istruzioni RISC originariamente progettato con lo scopo di scrivere programmi in un sottoinsieme di C che possono essere compilati in istruzioni BPF attraverso un back-end di compilatore (es. LLVM), in modo che il kernel possa successivamente mapparli, attraverso un compilatore JIT che si trova nel kernel, in opcode nativi, per migliorare le prestazioni di esecuzione all'interno del kernel. I registri eBPF e la maggior parte delle istruzioni vengono ora mappati uno a uno. Ciò facilita l'emissione di istruzioni eBPF da qualsiasi compilatore esterno (nello userspace). Ovviamente, prima di qualsiasi esecuzione, il bytecode generato viene passato attraverso un verifier nel kernel, per garantire l'integrità e evitare comportamenti anomali.

2.3.2 Kernel verifier

Esistono rischi di sicurezza e stabilità che consentono al codice dello spazio utente di essere eseguito all'interno del kernel [6]. Quindi, un certo numero di controlli vengono eseguiti su ogni programma eBPF prima che venga caricato. Il primo test garantisce che il programma eBPF termini e non contenga loop

che potrebbero causare il blocco del kernel[4]. Questo viene verificato facendo una ricerca in profondità del grafico del flusso di controllo[5] del programma (CFG). Le istruzioni irraggiungibili sono severamente proibite; qualsiasi programma contenente istruzioni non raggiungibili non verrà caricato.
Il secondo passo è più complesso e richiede al verificatore di simulare l'esecuzione del programma eBPF un' istruzione alla volta. Lo stato della macchina virtuale viene controllato prima e dopo l'esecuzione di ogni istruzione per garantire che lo stato del registro e dello stack siano validi. I salti fuori limite sono proibiti, così come l'accesso ai dati fuori dall'intervallo.
Inoltre il verificatore ha anche una "modalità sicura". L'idea è di assicurarsi che gli indirizzi del kernel non vadano persi dagli utenti non privilegiati e che i puntatori ai registri non possano essere scritti in memoria.

2.3.3 Mappe e strutture dati

Le principali strutture dati usate dai programmi eBPF sono le eBPF maps, generiche strutture che consentono di passare i dati avanti e indietro all'interno del kernel o meglio tra il kernel e lo spazio utente. Come suggerisce il nome "mappa", i dati vengono memorizzati e recuperati utilizzando una chiave [6].
Le mappe sono strutture dati efficienti di chiavi/valore che risiedono nello spazio del kernel. È possibile accedervi da un programma BPF e vengono utilizzate per mantenere persistente lo stato del programma BPF. Possono anche essere accessibili tramite descrittori di file dallo spazio utente e possono essere condivise arbitrariamente con altri programmi BPF o altre applicazioni definite in spazio utente. Inoltre non è necessario che i programmi BPF che condividono le stesse mappe siano dello stesso tipo, ad esempio i programmi di monitoraggio di sistema possono condividere le stesse mappe con i programmi per monitorare la rete. Esistono diversi tipi di mappe e ognuno offre un comportamento e un insieme di compromessi diversi:

[4]Inoltre non sarebbe performante utilizzare dei loop all'interno del codice eBPF perché rallenterebbero l'attività del kernel

[5]Il verificatore non ha bisogno di percorrere ogni percorso del programma, è abbastanza intelligente da sapere quando lo stato corrente del programma è un sottoinsieme di uno già controllato

- `BPF_MAP_TYPE_HASH`: semplice hash table;
- `BPF_MAP_TYPE_ARRAY`: un array-map, ottimizzata per ricerche veloci, spesso utilizzata per i contatori;
- `BPF_MAP_TYPE_PROG_ARRAY`: una serie di descrittori di file corrispondenti ai programmi eBPF; utilizzato per implementare tabelle di salto e sottoprogrammi per gestire specifici protocolli di pacchetti;
- `BPF_MAP_TYPE_PERCPU_ARRAY`: un per-CPU array, usato per istogrammi di latenza;
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`: memorizza i puntatori alla `struct perf_event`, usata per leggere e salvare i contatori per i perf events;
- `BPF_MAP_TYPE_CGROUP_ARRAY`: memorizza i puntatori dei gruppi di controllo;
- `BPF_MAP_TYPE_PERCPU_HASH`: una per-CPU hash table;
- `BPF_MAP_TYPE_LRU_HASH`: una hash table che conserva solo gli oggetti usati più di recente;
- `BPF_MAP_TYPE_STACK_TRACE`: memorizza le stack traces;
- `BPF_MAP_TYPE_SOCKET_MAP`: memorizza le socket e consente il reindirizzamento alle socket con le funzioni helper BPF.

Osservazione.
Un singolo programma BPF può attualmente accedere direttamente a 64 mappe diverse.

2.3.4 Helper

Le funzioni di supporto consentono ai programmi BPF di consultare un set di chiamate di funzioni definite nel kernel per recuperare e inviare dati da/verso il kernel. Le funzioni di supporto disponibili possono differire per ogni tipo di programma BPF, ad esempio, i programmi BPF collegati alle sockets possono solo chiamare un sottoinsieme di helper rispetto ai programmi BPF per il monitoraggio di sistema.

2.3.5 Tail calls

Un altro concetto che può essere utilizzato in BPF si chiama Tail calls. Le chiamate di coda possono essere viste come un meccanismo che consente ad un programma BPF di chiamarne un altro, senza tornare al vecchio programma, riutilizzando lo stack frame di quello precedente.

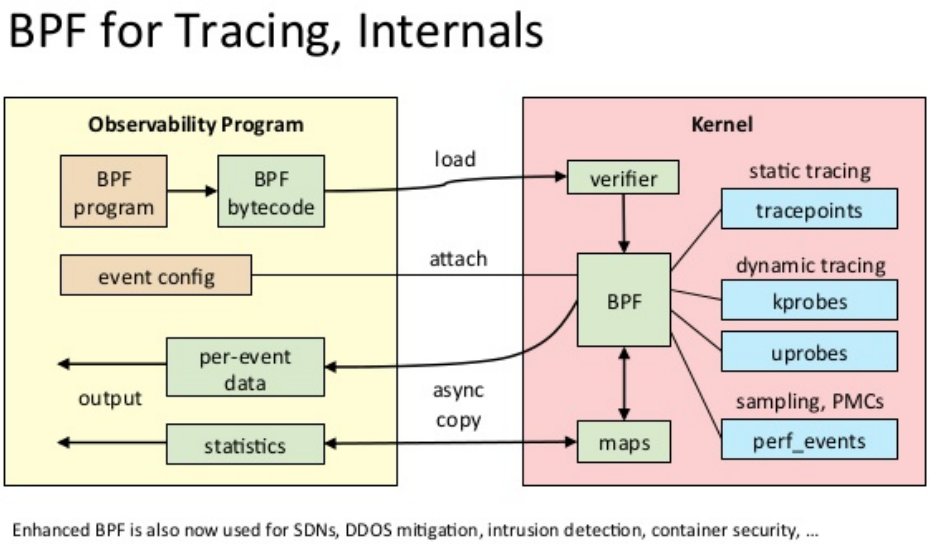


Figura 2.3: Architettura eBPF

La figura 2.3 mostra il workflow relativo all'utilizzo di un programma eBPF. Il codice viene compilato in un BPF bytecode e inviato al kernel, dove il verifier può rifiutarlo se lo ritiene pericoloso. Se il bytecode BPF viene accettato può essere collegato a diverse fonti di eventi:

- **kprobes** e **kret_probes**: kernel dynamic tracing;
- **uprobes** e **uret_probes**: user level dynamic tracing;
- **tracepoint**: kernel static tracing;
- **perf_events**: performance event.

Infine il programma BPF ha due modi per trasferire i dati misurati nello spazio dell'utente: fornendo i dettagli per ogni evento oppure tramite una

mappa BPF come avevamo descritto nella sezione 2.3.3.

Durante il periodo di prova dello strumento, ho sviluppato alcuni tool per imparare ad usare eBPF, prima di poterlo utilizzare per la costruzione di strumenti adatti all'analisi del traffico di rete.
Il seguente esempio, da me progettato, viene presentato per illustrare i passi necessari relativi allo sviluppo di un programma eBPF. In particolare vedremo come si costruiscono i moduli (kernel agent) che successivamente verranno caricati nello spazio del kernel, e come avviene l'interazione tra quest'ultimi e l'applicazione in spazio utente, che si occuperà di stampare a video le informazioni relative al processo di monitoraggio.

Esempio di programma eBPF: `cdsnoop`

Il seguente esempio mostra il funzionamento di un programma eBPF da me chiamato `cdsnoop`, il cui utilizzo permette di monitorare tutti i processi bash che fanno uso del comando `cd` per spostarsi tra le directory del file system di una macchina con sistema operativo Linux.

Per prima cosa bisogna andare a definire il funzionamento del kernel agent che in questo caso deve:

1. Leggere l'input da bash, ascoltando l'evento della bash `readline`;
2. Controllare se sono stati riscontrati errori nell'eseguire l'input della bash, ascoltando l'evento `get_name_for_error`;
3. Sapere il nome della directory corrente prima dell'esecuzione del comando cd, ascoltando l'evento `get_string_value`.

Mentre il processo in spazio utente si occupa di filtrare le informazioni che riceve dai kernel agent e di formattare l'output in base all'evento.
Le righe seguenti mostrano il funzionamento in linee generali di `cdsnoop`:

```
#!/usr/bin/env python
from __future__ import print_function
from bcc import BPF
from time import strftime

'''program eBPF'''
bpf_text = '''
#include <uapi/linux/ptrace.h>

int printret(struct pt_regs *ctx) {
    if (!ctx->ax )
        return 0;
    char str[30] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->ax);
    bpf_trace_printk("%s\\n", &str);
    return 0;
}

[...]'''

'''load BPF program'''
b = BPF(text=bpf_text)
'''cd error probe'''
b.attach_uretprobe(name="/bin/bash",
     sym="get_name_for_error",fn_name="printerr")
'''bash readline probe'''
b.attach_uretprobe(name="/bin/bash", sym="readline",
     fn_name="printret")
'''get current directory probe'''
b.attach_uretprobe(name="/bin/bash",
    sym="get_string_value",fn_name="printcurrDir")
'''header'''
print("%-12s %-10s %-15s %-10s %s" % ("TIME", "PID",
    "PREV_DIR", "CURR_DIR" ,"COMM"))
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
            continue
    [...]
```

Il programma eBPF viene dichiarato come stringa: righe 7-19; dopodiché viene caricato nella riga 22; per poi essere associato a 3 diverse fonti di eventi[6]. Il gestore dell'evento `readline` è la funzione `printret`, che una volta rilevato l'evento scrive sul trace_pipe il valore contenuto nell'input della bash, che come abbiamo visto nella sezione 2.3.1 può essere recuperato leggendo il contenuto del registro `ctx->ax`. Successivamente il processo a livello utente può ricevere le informazioni mandate dal kernel agent via trace_pipe utilizzando il comando: `(task, pid, cpu, flags, ts, msg) = b.trace_fields()` che permette di recuperare il nome del task, il pid del processo, le statistiche sulla cpu, i flag associati a quell'evento, il timestamp e il contenuto del registro `ctx->ax` che si trova nel campo msg.
Nel resto del codice il processo in spazio utente controlla che non ci siano errori (recuperando le statistiche del kernel agent che ascolta l'evento `get_string_value`), recupera la directory corrente (quella prima dell'esecuzione del comando `cd`) e stampa a video le statistiche come in figura 2.4.

```
root@riccardo-Lenovo:~/Scrivania# ./cdsnoop.py
TIME         PID        PREV_DIR        CURR_DIR   COMM
14:22:52     4254       riccardo        Scrivania  cd Scrivania/
14:23:03     4254       Scrivania       bin        cd /usr/bin/
14:23:12     4296       riccardo        home       cd ..
14:23:14     4296       home            /          cd ..
14:23:28     4296       /               Scrivania  cd /home/riccardo/Scrivania/
14:23:33     4296  bash : cd : djasiodjasio File o directory non esistente
14:23:37     4254  bash : cd : dasaskdop File o directory non esistente
14:23:38     4254       bin             usr        cd ..
14:23:40     4254       usr             /          cd ..
14:23:46     4254       /               home       cd home/
```

Figura 2.4: Statistiche cdsnoop

A differenza degli strumenti di monitoraggio presentati nelle sezioni precedenti le considerazioni finali su eBPF verranno presentate nei capitoli successivi.

[6]Le funzioni `printerr, printcurrDir` non sono riportate nel codice perchè hanno lo stesso comportamento della funzione `printret` definita nelle righe 10-16

Capitolo 3

Progetto

In questo capitolo si parlerà di come eBPF migliora l'analisi del traffico di rete, attraverso l'esecuzione di alcuni tool, da me sviluppati, collegati ad eventi Linux.

Nella sezione 2.3.1 abbiamo analizzato l'architettura generale di eBPF, che comprende un set di istruzioni RISC, un verificatore nel kernel per la corretta esecuzione dei programmi, le strutture dati principali, funzioni di supporto e tail calls. Ad oggi con l'avanzare delle versioni del kernel, si è andata a ridefinire in modo sempre più dettagliato l'architettura di eBPF aggiungendo un ulteriore infrastruttura attorno ad esso che fa uso di tecniche per la rilevazione degli eventi che possono presentarsi in sistemi Linux. La figura 3.1 mostra in linee generali questi eventi[1] e le funzionalità di supporto messe a disposizione da eBPF contrassegnate dalla versione del kernel in verde.
I tool, da me sviluppati, che verranno presentati in questo capitolo consistono nell'esecuzione di mini programmi costituiti da due componenti, un insieme di kernel agent e un processo in spazio utente. I kernel agents operano nello spazio del kernel e usano dei meccanismi orientati agli eventi per monitorare i cambiamenti dello stato del sistema e della rete causati dai processi in esecuzione; mentre il processo in spazio utente, preleva quindi gli eventi sui quali era in ascolto il kernel agent.

[1]Per una lista dettagliata di tutti gli eventi Linux vedere: /sys/kernel/debug/tracing

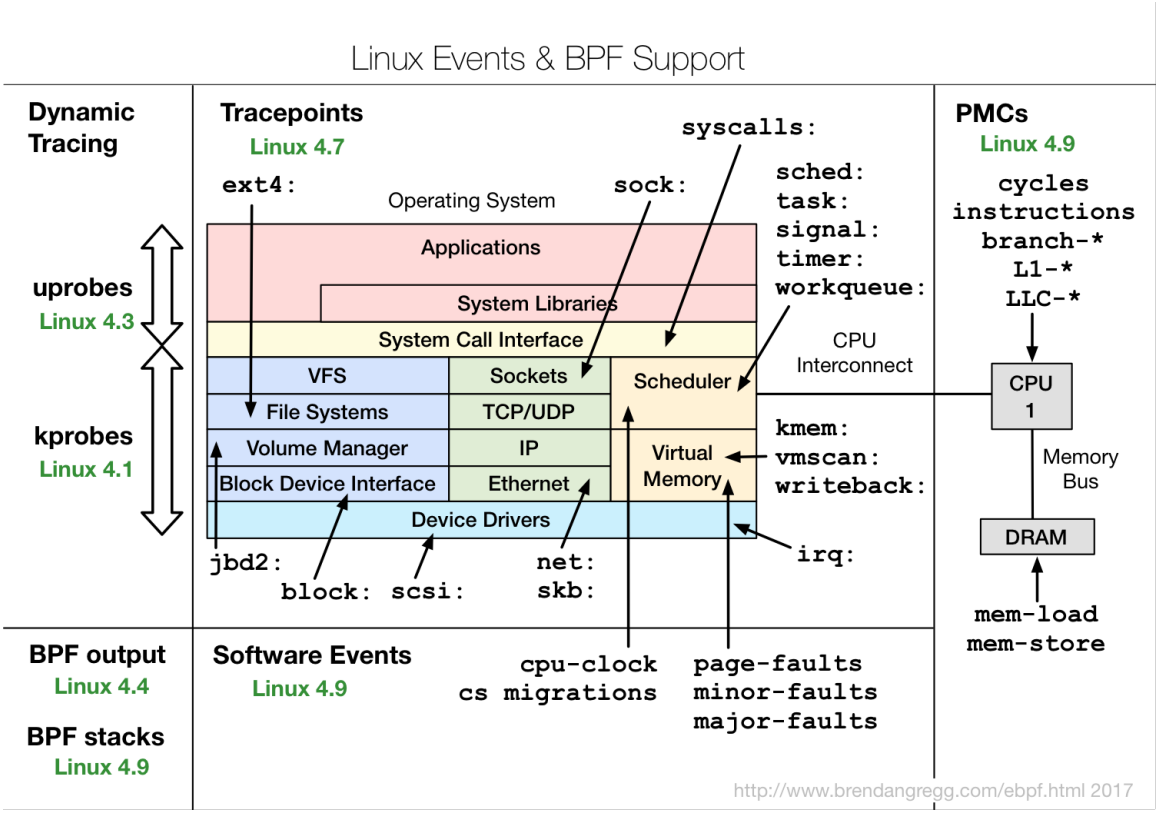


Figura 3.1: Eventi Linux e supporto eBPF

3.1 Prerequisiti

Prima di presentare i tool sviluppati nelle sezioni che seguiranno vengono analizzati i prerequisiti e gli strumenti necessari per la creazione di un programma eBPF.

3.1.1 Kernel version

Come abbiamo visto nella sezione 1.1, nelle ultime versioni del kernel sono stati aggiunti all'interno del codice i così detti tracepoint. In generale per usare eBPF è richiesta una versione del kernel Linux superiore o uguale alla 4.1 in modo da poter sfruttare i tracepoint importati nel codice kernel.

Riprendendo la figura 3.1 si può notare che versioni più recenti del kernel favoriscono una vista più ampia e dettagliata per il monitoraggio del sistema e della rete.

3.1.2 BPF Compiler Collection (Bcc)

Mentre eBPF offre grandi potenzialità, c'è un problema: è difficile da configurare tramite il suo assembly o attraverso un interfaccia C. Nasce allora un progetto chiamato BPF Compiler Collection (bcc).
Bcc [9] è toolkit che fa uso di eBPF, mette a disposizione una libreria utilizzata per la manipolazione del kernel e include diversi strumenti ed esempi utili. La maggior parte degli strumenti che usa bcc richiede Linux 4.1 e/o versioni successive. Bcc semplifica la scrittura di programmi eBPF, grazie alla strumentazione del kernel programmabile in C (include un wrapper C attorno a LLVM) e all'utilizzo di interfacce front-end in Python e Lua. È adatto a molte attività, tra cui l'analisi delle prestazioni e il controllo del traffico di rete.
Grazie a questa interfaccia possiamo creare strumenti ad hoc in grado di monitorare le performance di sistemi Linux. I tool che verranno presentati utilizzano come interfaccia front-end Python.

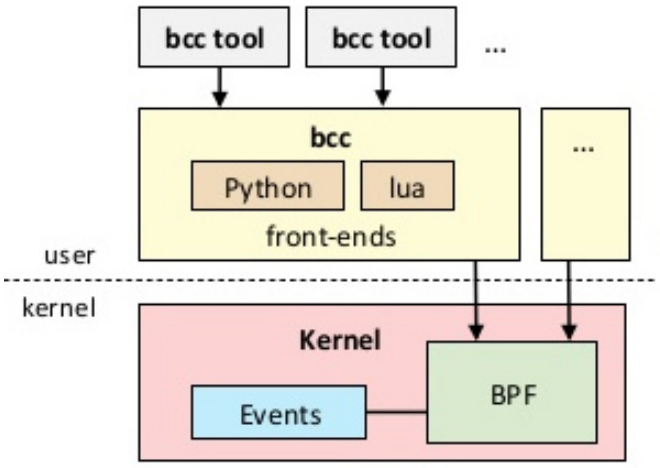


Figura 3.2: Supporto Bcc

3.1.3 Restrizioni sul kernel agent

Per poter sviluppare dei tool per il monitoraggio sia di sistema che di rete è necessario tenere in considerazione che:

- **eBPF è C-restricted**: non si può usare nessun loop o syscall all'interno di un programma eBPF. Si possono solo usare funzioni `bpf_*`;
- **Tutta la memoria deve essere letta sullo stack BPF prima di essere manipolata** attraverso `bpf_probe_read()`, che esegue i controlli necessari per il corretto funzionamento del programma.

Inoltre le uniche soluzioni per spedire i dati dal kernel agent al processo in spazio utente sono:

- `bpf_trace_printk()`: utilizzato per il debug scrive su trace_pipe e può scontrarsi con altri programmi di monitoring, sono quindi consigliati i metodi che seguono:
- `BPF_HISTOGRAM()` o altri tipi di mappe in riferimento alla sezione 2.3.3.
- `BPF_PERF_OUTPUT()`: è un modo per mandare i dettagli del traffico per-evento allo spazio utente, tramite una struttura dati personalizzata dall'utente. Il programma python (spazio utente) deve quindi avere un modulo, in questo caso `ct` per la definizione di struct come quelle del C.

3.2 Primo tool del progetto: `TcpLatency`

Il primo tool che verrà presentato è `TcpLatency` che viene utilizzato per misurare e analizzare la latenza di rete, la quale gioca un ruolo molto importante nelle reti moderne. Monitorare la latenza è fondamentale per garantire una buona qualità del servizio (QoS) soprattutto nelle applicazioni interattive. Una latenza eccessiva crea, all'interno delle reti, colli di bottiglia che causano la riduzione di banda effettiva. Per questa ragione quando si determina la velocità di rete è importante eseguire dei test sulla latenza della rete.
I tool esistenti monitorano la latenza di rete in modo molto approssimativo, possono generare molto overhead sulla CPU, oppure come spesso accade non hanno la capacità di monitorarla in ambienti containerizzati.
`TcpLatency` utilizza flussi richiesta/risposta TCP per monitorare la latenza di rete con eBPF. Nello specifico il tool sviluppato richiede i seguenti requisiti:

- **Analizzare il tempo di risposta della rete**: per vedere la latenza di rete, calcolata utilizzando un kernel agent che ascolta l'evento `tcp_v4/v6_connect` e calcola i tempi di andata/ritorno necessari per stabilire una connessione;

- **Identificare il processo che riduce le performance di rete**: con un test sulla latenza di rete, gli analisti della rete possono identificare l'origine e la natura dei problemi di affidabilità e di performance delle reti o delle applicazioni conoscendo il pid del processo che ha causato il rallentamento;

- **Vedere se i problemi sono causati dalle app o dalla rete**: analizzando e identificando il traffico di rete si può determinare se il rallentamento di comunicazione su un collegamento è causato dalle applicazioni o se il problema proviene dalla rete (ad esempio congestione nella rete).

Nella figura qui accanto viene mostrato il lavoro del tool `TcpLatency` in uno tipico scenario client/server. Il tool, in questo caso è in continuo ascolto sull'evento `tcp_v4/v6_connect` e sul suo ritorno (`tcp_rcv_state_process`).

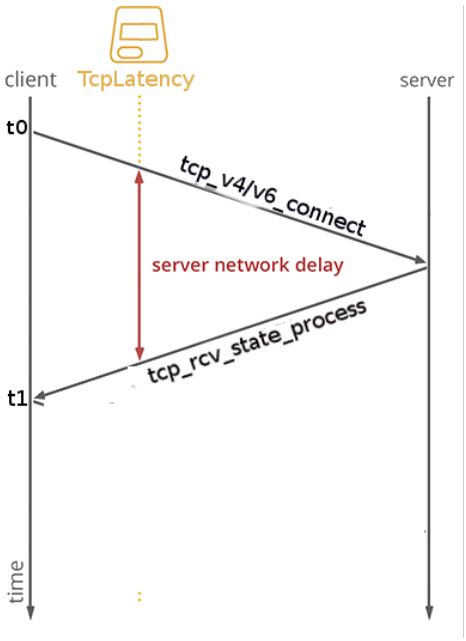


Figura 3.3: TcpLatency tool

Ogni qual volta che un'applicazione o un processo in spazio utente, invia una richiesta di connessione (`socket.connect()`) verso un server, `TcpLatency` memorizza in una `BPF_TABLE_HASH` le informazioni del processo, quali pid (che in questo caso sarà la chiave della tabella), nome del task, il tempo nel quale è avvenuta la chiamata (t_0) e la socket sulla quale è stata mandata la richiesta di connessione (che sarà il valore corrispondente al pid). Al momento del ritorno, tempo $t1$, il tool verifica se la richiesta è andata a buon fine, controllando il flag dello stato della connessione TCP relativo alla socket sulla quale si era fatto la richiesta, in caso di successo (`sk_state = TCP_ESTABLISHED`) recupera dalla tabella hash le informazioni del processo e la rispettiva socket, dalla quale preleva la quadrupla [source_ip_address ; destination_ip_address ; source_TCP_port ; destination_TCP_port], calcola:

$$\Delta t = t_1 - t_0$$

e manda al processo in spazio utente le statistiche dell'evento processato utilizzando un BPF_PERF_OUTPUT() buffer.
L'architettura dello strumento coinvolge quindi due kenel agent, che condividono la stessa hashmap, e sono collegati a due diverse fonti di eventi:

- `kprobe(event = "tcp_v4/v6_connect")`: per la richiesta di connessione;
- `kret_probe(event = "tcp_rcv_state_process")`: per il controllo sul-

lo stato della connessione TCP.

I gestori di questi due eventi, come si può immaginare, hanno una struttura minimale, in quanto devono solamente leggere informazioni dalle strutture dati coinvolte, come le socket e la tabella hash che condividono.

3.3 Secondo tool del progetto: `NetPagefault`

In questa sezione viene presentato il secondo tool `NetPagefault`, che permette di monitorare i processi che utilizzano la rete e sapere se questi ultimi causano pagefault in memoria.
Monitorare le applicazioni che comunicano attraverso il web diventa fondamentale per la gestione della rete. Il tool di monitoraggio in questione richiede i seguenti requisiti:

- **Informazioni Real-Time**: statistiche accurate e tempestive sulle risorse di rete;
- **Correzione di problemi**: se ci sono dei fallimenti o irregolarità possiamo intervenire subito;
- **Basso overhead**: il sovraccarico della rete deve essere ridotto al minimo per garantire che l'analisi delle informazioni si svolga in modo efficiente.

Questi sono i requisiti necessari per i tool che monitorano i processi che utilizzano la rete. Ma cosa succede se questi processi oltre che a sovraccaricare la rete rallentano anche il sistema operativo, causando errori di paginazione? In particolare, cosa succede quando il kernel non ha più memoria fisica per gestire le pagine richieste da questi processi? In condizioni normali il kernel di Linux gestisce le pagine di memoria in modo che lo spazio degli indirizzi virtuali sia mappato sulla memoria fisica e ogni processo abbia accesso ai dati e al codice di cui ha bisogno. In mancanza di memoria per gestire le pagina vorremmo che il sistema continuasse a funzionare. Il kernel, in questi casi, inizierà a scrivere sul disco alcune delle pagine che tiene in memoria e userà le pagine appena liberate per soddisfare gli errori di pagina correnti.

Ma la scrittura di pagine su disco è relativamente lenta rispetto alla velocità della CPU e della memoria principale. Sono quindi necessarie soluzioni che individuano e gestiscono tali anomalie senza ridurre velocità e le prestazioni, le quali giocano un ruolo molto importante per l'esperienza dell'utente che utilizza applicazioni network come Web browser, riproduttori audio/video, server cloud, ecc.
`NetPagefault` permette di individuare quali sono le applicazioni che influiscono sulle performance del nostro pc, combinando i feedback del sistema operativo con quelli ottenuti analizzando le applicazioni che usano la rete. Il tool sviluppato monitora tutti i processi che usano i protocolli TCP e UDP a livello trasporto, sia lato client che lato server, e controlla i pagefault causati da essi. In particolare lo strumento qui presentato è in continuo ascolto sulle connessioni TCP lato client monitorando l'evento `tcp_v4_connect` e il relativo ritorno, e lato server monitorando il ritorno dell'evento `inet_csk_accept` che corrisponde quindi alla classica chiamata `new_socket = socket.accept()`. Per quel che riguarda il protocollo UDP, essendo un protocollo connectionless si devono monitorare gli eventi `udp_sendmsg` lato server e `udp_rcvmsg` lato client. L'architettura dello strumento coinvolge quindi tre kernel agent:

1. **TCP_tracerAgent** che ascolta gli eventi:
   - `kprobe(event = "tcp_v4_connect");`
   - `kret_probe(event = "tcp_v4_connect");`
   - `kret_probe(event = "inet_csk_accept").`
2. **UDP_tracerAgent** che ascolta gli eventi:
   - `kprobe(event = "udp_sendmsg");`
   - `kret_probe(event = "udp_sendmsg");`
   - `kret_probe(event = "udp_rcvmsg").`
3. **Fault_tracerAgent** che ascolta l'evento:
   - `perf_event(ev_type=PerfType.SOFTWARE, ev_config=PerfSWConfig.PAGE_FAULTS_MIN).`

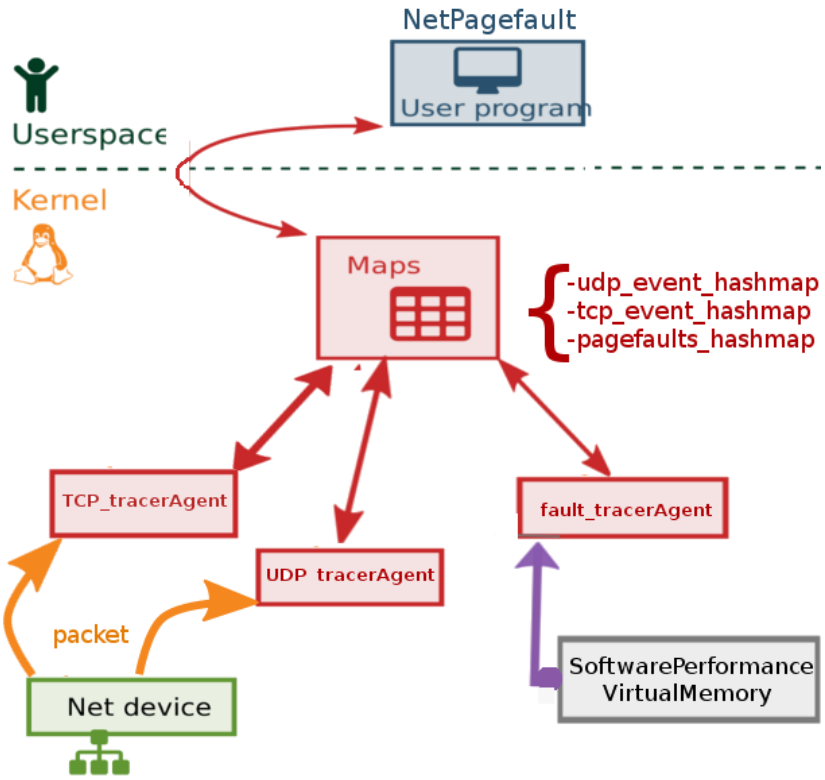


Figura 3.4: Architettura di NetPagefault

La figura 3.4 mostra l'architettura di `NetPagefault`. I due kernel agent **TCP_tracerAgent** e **UDP_tracerAgent** recuperano le informazioni dai Network device, mentre **Fault_tracerAgent** recupera le informazioni relative ai pagafault direttamente della memoria virtuale di Linux. Tutti e tre i kernel agent inseriscono i propri report nelle rispettive tabelle hash, in modo da permettere al processo utente di poter recuperare, e filtrare se necessario, le informazioni processate dai tre componenti.

Capitolo 4

Validazione e risultati

In questo capitolo verranno presentati i risultati finali e la validazione dei tool `NetPagefault` e `TcpLatency` a partire dai requisiti presentati nel capitolo precedente.
La validazione in [4] è un'attività di controllo mirata a confrontare il risultato di una fase del processo di sviluppo con i requisiti del prodotto; tipicamente con quanto stabilito dal contratto o, meglio, dal documento di analisi dei requisiti. Un comune esempio di validazione è il controllo che il prodotto finito abbia funzionalità e prestazioni conformi con quelle stabilite all'inizio del processo di sviluppo. Inoltre limitandosi ad aspetti particolari, è possibile effettuare delle operazioni di validazione anche durante il processo di sviluppo, testando separatamente i moduli del sistema generale; sfortunatamente, per via della sua architettura, eBPF non permette la scomposizione di programmi in moduli separati, perciò per validare i tool sviluppati eseguiremo solo test sul sistema e non su singole componenti.
Alla fine del processo di validazione, analizzeremo la qualità degli strumenti `NetPagefault` e `TcpLatency`, rispetto ai requisiti di efficienza che possono essere controllati solo a fronte della messa in esecuzione del software.
Tutti i test sono stati eseguiti su una macchina con processore AMD E1-6010 APU - 1.35GHz e sistema operativo Ubuntu 16.04 LTS con versione del kernel 4.13.0-37-generic.

4.1 Test sul sistema di `TcpLatency`

Il test di sistema è la più canonica delle attività di validazione che valuta ogni caratteristica di qualità del prodotto software nella sua completezza, avendo come documento di riscontro i requisiti dell'utente, che nel caso di `TcpLatency`, sono:

1. Facilità d'uso del sistema da parte di utenti finali (*usabilità*);
2. calcolo del lasso di tempo di andata e ritorno di un pacchetto che viaggia lungo un percorso della rete che coinvolge un mittente e un destinatario (*funzionalità*);
3. valori accurati e realistici sul tempo (misurato in ms), sui processi e sulle informazioni necessarie per identificare una specifica connessione: la quadrupla (ip_src, ip_dest, port_src, port_dest) (*funzionalità*);
4. statistiche real-time, i valori del punto precedente devono essere riportati all'utente in tempo reale senza rallentamenti (*efficienza*).

Le tecniche più adottate per i test sul sistema sono basate su criteri funzionali. Gli obiettivi dei controlli di sistema sono normalmente mirati a esercitare il sistema sotto ben determinati aspetti. Nelle sezioni che seguiranno analizzeremo i criteri sui quali è stata testata la validità di `TcpLatency`.

4.1.1 Test sull'usabilità

Con questo controllo si riesce a valutare la facilità d'uso del tool da parte dell'utente che vuole monitorare la latenza di comunicazione nella rete. In particolare una volta avviato il tool l'utente può conoscere la latenza relativa ad una specifica connessione, utilizzando l'applicazione che gli permette di accedere a quel determinato servizio; ad esempio se si vuole conoscere la latenza di comunicazione di un server web, basterà utilizzare un browser qualunque e recarsi sulla pagina del server, in questo caso `TcpLatency` riconoscerà il processo (il browser in questo caso) e invierà le statistiche relative alla latenza del collegamento Utente_mittente - Server_web.

4.1.2 Test sulle funzionalità

È il più intuitivo dei controlli, quello cioè che mira a controllare che ogni funzionalità del prodotto stabilita nei requisiti sia stata realizzata correttamente. Con questo tipo di test sono stati controllati i requisiti 2 e 3 della sezione 4.1.
Per quel che riguarda quindi i valori dei tempi di latenza, ovvero il Δt per l'andata e ritorno di un pacchetto, sono stati calcolati utilizzando un helper (`bpf_ktime_get_ns()`) messo a disposizione da eBPF che consente di ottenere dei valori accurati del tempo, espresso in nanosecondi, da clocksource CLOCK_MONOLITIC. Mentre i valori delle informazioni del processo che comunica in rete, sono ricavabili dai due helper: `bpf_get_current_pid_tgid()` e `bpf_get_current_comm(&task, sizeof(task))` per ricavare rispettivamente il pid del processo e il suo nome simbolico. In ultimo le informazioni della quadrupla [ip_src; ip_dest; port_src; port_dest] sono ricavabili dalla socket relativa alla comunicazione accedendo ai campi: [`skp->sk_rcv_saddr; skp->sk_daddr; skp->sk_num; skp->sk_dport`].
I requisiti funzionali sono quindi soddisfatti dagli helper della libreria di eBPF, grazie alla loro stabilità permettono di ricavare le informazione relative agli eventi monitorati in `TcpLatency`; inoltre come vedremo nella prossima sezione riescono anche a garantire buone performance.

4.1.3 Test sull'efficienza

I test che sono stati eseguiti sullo strumento riguardano in modo particolare le performance del sistema sotto un punto di vista della velocità e della gestione di risorse dello strumento, che sono proprietà necessarie soprattutto in corrispondenza di carichi eccessivi nella rete.
Quando si parla di velocità si intende ricevere le informazioni del monitoraggio in tempi più brevi possibile. Vengono quindi aggiunti dei requisiti che riguardano l'efficienza dei tempi di risposta dello strumento. Grazie alle componenti di `TcpLatency` che si trovano nel kernel, si possono ricavare informazioni in tempo reale senza avere rallentamenti; questo perché l'esecuzione dei kernel agent dello strumento è rapidissima per via della loro struttura minimale, quindi, l'esecuzione vera e propria del kernel viene interrotta per

un tempo brevissimo e l'utente non subisce rallentamenti anche in presenza di molto carico nella rete.
Un altro aspetto da non sottovalutare è la gestione delle risorse, soprattutto in eccessiva quantità di carico nella rete. Come descritto nella sezione 3.2 del capitolo precedente, `TcpLatency` utilizza una hashmap per tener traccia dei processi che fanno una richiesta di connessione per testare la latenza. Le informazioni dei processi, compresa la socket sulla quale è stata inviata la richiesta di connessione, vengono salvate all'interno della tabella, in modo da poterle recuperare in caso di stabilimento della connessione. Tuttavia cosa succede se si aggiunge carico alla rete e le entry della tabella si saturano? Al ritorno della connessione `TcpLatency`, sia in caso di successo che di errore deve necessariamente eliminare le informazioni corrispondenti a quella connessione in modo da riuscire a liberare le entry della tabella, alla fine di ogni evento.

4.2 Risultati e considerazioni finali su `TcpLatency`

Dopo aver analizzato i requisiti e testato le funzionalità del tool di monitoraggio per la latenza di rete, possiamo adesso presentare i risultati finali di `TcpLatency`, mettendolo a paragone con il comando bash `ping`, che utilizza il protocollo ICMP per conoscere l'intervallo di tempo che intercorre tra l'invio di una `ICMP ECHO_REQUEST` e l'arrivo di una `ICMP ECHO_RESPONSE` da parte dell'host sul quale stiamo testando la latenza.
A differenza di `TcpLatency`, che come abbiamo descritto nella sezione 3.2 ha il compito di calcolare la latenza ascoltando gli eventi del kernel, il `ping` deve ricevere in input l'indirizzo IP del server web o di un suo alias per poter mandare un pacchetto ICMP e calcolare la latenza di rete.

Nel seguente scenario vengono mostrati i risultati del calcolo dei tempi di latenza da parte dei due tool, relativi alle connessioni: amazon.it, youtube.com e localhost. `TcpLatency` in questo caso utilizza un browser per calcolare la latenza di rete relativa ai due server web e il comando `ssh` per l'host locale; come si può vedere dalla figura 4.1 i tempi di latenza calcolati dai due tool non differiscono di molto.

```
riccardo@riccardo-Lenovo:~$ ping amazon.it
PING amazon.it (52.95.116.114) 56(84) bytes of data.
64 bytes from 52.95.116.114: icmp_seq=1 ttl=230 time=56.0 ms
64 bytes from 52.95.116.114: icmp_seq=2 ttl=230 time=51.7 ms
64 bytes from 52.95.116.114: icmp_seq=3 ttl=230 time=50.9 ms
64 bytes from 52.95.116.114: icmp_seq=4 ttl=230 time=53.6 ms
^C
--- amazon.it ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 50.933/53.088/56.094/2.004 ms
riccardo@riccardo-Lenovo:~$ ping youtube.com
PING youtube.com (216.58.205.110) 56(84) bytes of data.
64 bytes from mil04s26-in-f110.1e100.net (216.58.205.110): icmp_seq=1 ttl=51 tim
e=22.8 ms
64 bytes from mil04s26-in-f110.1e100.net (216.58.205.110): icmp_seq=2 ttl=51 tim
e=21.2 ms
64 bytes from mil04s26-in-f110.1e100.net (216.58.205.110): icmp_seq=3 ttl=51 tim
e=19.1 ms
64 bytes from mil04s26-in-f110.1e100.net (216.58.205.110): icmp_seq=4 ttl=51 tim
e=18.7 ms
64 bytes from mil04s26-in-f110.1e100.net (216.58.205.110): icmp_seq=5 ttl=51 tim
e=18.8 ms
^C
--- youtube.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 18.767/20.188/22.899/1.647 ms
riccardo@riccardo-Lenovo:~$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.105 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.122 ms
^C
--- localhost ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1021ms
rtt min/avg/max/mdev = 0.105/0.113/0.122/0.013 ms
```

(a) Ping command

```
root@riccardo-Lenovo:~/Scrivania/Tesi/eBPF_Program/NetLatency# ./tcp_latency.py
PID    COMM             SADDR           DADDR           DPORT SPORT LAT(ms)
5625   Chrome_IOThread 192.168.1.84    52.94.220.16    443   50818 52.416000
5625   Chrome_IOThread 192.168.1.84    52.94.220.16    443   50820 52.405000
5625   Chrome_IOThread 192.168.1.84    52.94.220.16    443   50822 54.710000
5625   Chrome_IOThread 192.168.1.84    54.239.36.249   443   39270 51.648000
5625   Chrome_IOThread 192.168.1.84    54.239.36.249   443   39272 52.406000
5625   Chrome_IOThread 192.168.1.84    216.58.205.98   443   53214 19.633000
5625   Chrome_IOThread 192.168.1.84    216.58.205.98   443   53216 20.027000
2960   dropbox         192.168.1.84    162.125.34.137  443   46030 211.551000
28258  ssh             127.0.0.1       127.0.0.1       22    34290 0.260000
```

(b) TcpLatency tool

Figura 4.1: Confronto tra `ping` e `TcpLatency`

Ci sono invece delle situazioni in cui il `ping` non riesce a trovare la porta relativa ad una specifica connessione, e quindi in questi casi non può calcolare la latenza di rete. Nella figura 4.2, viene mostrato questo tipo di scenario, in cui il `ping` non riceve il pacchetto `ICMP ECHO_RESPOSE` dal server di.unipi.it e di conseguenza non riesce a calcolare la latenza, mentre `TcpLatency`, connettendosi al server web attraverso il browser, riesce a calcolare i tempi di latenza della connessione.

```
riccardo@riccardo-Lenovo:~$ ping di.unipi.it
PING di.unipi.it (131.114.3.24) 56(84) bytes of data.
From www.di.unipi.it (131.114.3.24) icmp_seq=1 Destination Port Unreachable
From www.di.unipi.it (131.114.3.24) icmp_seq=2 Destination Port Unreachable
^C
--- di.unipi.it ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1001ms
```

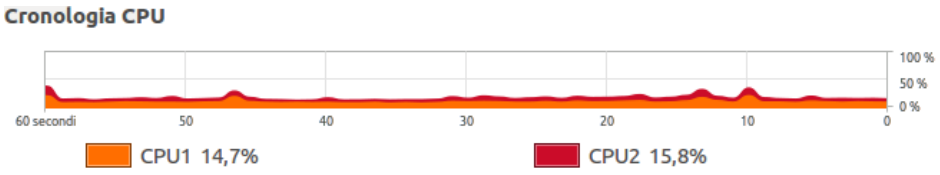
(a) Ping command

```
t@riccardo-Lenovo:~/Scrivania/Tesi/eBPF_Program/NetLatency# ./tcp_latencpy
 COMM            SADDR           DADDR           DPORT SPORT LAT(ms)
 Chrome_IOThread 192.168.1.84    131.114.3.24    443   41058 15.447000
 Chrome_IOThread 192.168.1.84    131.114.3.24    443   41060 15.341000
 Chrome_IOThread 192.168.1.84    131.114.3.24    443   41062 15.179000
```

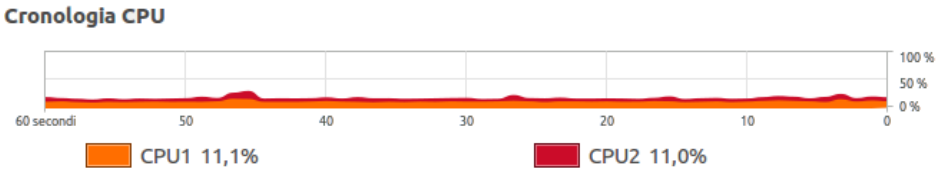
(b) TcpLatency tool

Figura 4.2: Confronto tra `ping` e `TcpLatency`

I risultati ottenuti da `TcpLatency` in questi casi risultano più efficienti, rispetto a quelli ottenuti dal comando `ping` preso come strumento di confronto durante la fase di testing. Vediamo il carico sulla CPU da parte dei due strumenti.



(a) Statistiche CPU `ping` in esecuzione



(b) Statistiche CPU `TcpLatency` in esecuzione

Figura 4.3: Grafici relativi alla cronologia della CPU in presenza del comando `ping` (a) e in presenza (b) di `TcpLatency`

Come si può vedere dalla figura 4.3 entrambi gli strumenti non aggiungono molto carico sulla CPU, in particolare, con lo strumento da me sviluppato, si riesce ad diminuire, anche se di poco, il lavoro della CPU, rispetto al comando `ping`. Possiamo quindi concludere che lo strumento `TcpLatency` è efficiente in termini di utilizzo della CPU.

4.3 Test sul sistema di `NetPagefault`

Nelle sezioni precedenti sono state analizzate le funzionalità dello strumento `TcpLatency` a partire dall'analisi dei requisiti necessari per il corretto funzionamento del sistema; in queste sezioni analizzeremo, invece, i criteri sui quali è stata testata la validità del tool `NetPagefault`, avendo come documento di riscontro i seguenti requisiti:

1. Facilità d'uso del sistema da parte di utenti finali (*usabilità*);
2. rilevare tutte le applicazioni che fanno uso della rete, e classificarle in base al protocollo[1] (*funzionalità*);
3. classificare i processi sia lato client che lato server (*funzionalità*);
4. monitorare il numero di pagefault causati in memoria dai processi che utilizzano la rete (*funzionalità*);
5. statistiche accurate, sui processi e informazioni sulle connessioni (*efficienza*).

4.3.1 Test sull'usabilità

`NetPagefault`, così come `TcpLatency`, deve rispettare il requisito di usabilità, quindi deve essere facile da usare per un utente che vuole monitorare le comunicazioni nella rete. Il tool deve essere avviato dall'utente inserendo come input l'intervallo di tempo necessario per il monitoraggio, una volta avviato permette di monitorare le connessioni TCP e UDP, sia lato client che lato server, e i pagafault dei processi che utilizzano i due protocolli, mentre l'utente sta svolgendo le proprie attività. Allo scadere del tempo vengono stampate le informazioni del monitoraggio direttamente a video, al contrario di come avveniva in `TcpLatency` che stampava le informazioni volta per volta.

[1] NetPagefault classifica solamente i processi che fanno uso dei protocolli TCP e UDP.

4.3.2 Test sulle funzionalità

Con questo tipo di test sono stati controllati i requisiti 2, 3 e 4 della sezione 4.3.
Per quel che riguarda i requisiti 2 e 3, le applicazioni che soddisfano questi criteri sono tutte quelle che utilizzano connessioni al di fuori del sistema operativo, utilizzando quindi socket AF_INET; nello specifico i requisiti vengono soddisfatti grazie all'uso dei kernel agent descritti nella sezione 3.3 (`TCP_tracerAgent` e `UDP_tracerAgent`) che leggono dallo stack di eBPF il tipo di socket, sulla quale è stato rilevato l'evento, attraverso il comando `bpf_probe_read(&family,sizeof(family),&sk->__sk_common.skc_family)`.
Mentre la classificazione dei protocolli viene fatta al momento della dichiarazione di `TCP/UDP_tracerAgent`, bisogna che entrambi i kernel agent, possano distinguere i tipi di connessione, sia lato client che lato server; il requisito in questo caso è soddisfatto dal supporto di eBPF che consente ai due kernel agent di poter ascoltare gli eventi: `tcp_v4_connect` , `udp_sendmsg` lato client e `inet_csk_accept`, `udp_rcvmsg` lato server.
Infine per soddisfare il requisito 4, `Fault_tracerAgent`, il kernel agent che ascolta gli eventi che riguardano le performance del sistema operativo, deve recuperare le informazioni relative ai fault in memoria causati dai processi che utilizzano la rete e inserirle in una tabella hash. Quando avviene un page fault in memoria, il kernel agent recupera le informazioni del processo, utilizzando gli helper già discussi nella sezione 4.1.2, e incrementa il numero di page fault causati da quel processo in base al contenuto della `struct bpf_perf_event_data *ctx`, che permette di recuperare il numero di errori di paginazione attraverso il campo `ctx->sample_period` relativo all'evento che si sta monitorando.
I requisiti funzionali, così come per lo strumento precedente, vengono soddisfatti grazie al supporto degli helper di eBPF.

4.3.3 Test sull'efficienza

I test fondamentali, effettuati sullo strumento, riguardano le prestazioni in termini di efficienza dello strumento. Le statistiche del monitoraggio di `NetPagefault` devono essere accurate e affidabili, inoltre bisogna aggiungere un ulteriore requisito allo strumento per far si che l'utente finale possa monitorare le proprie applicazioni durante le proprie attività senza avere rallentamenti.
Anche in questo caso, come per `TcpLatency`, l'efficienza dello strumento è data dalla struttura minimale dei kernel agent, che riescono a garantire buone prestazioni anche in presenza di molto stress nella rete e/o nel sistema operativo.

4.4 Risultati e considerazioni finali su `NetPagefault`

In questa sezione vengono presentati i risultati finali di `NetPagefault`, mettendolo a paragone con il comando bash `top`, confrontando i risultati ottenuti con il numero di pagefault osservati dallo strumento, e i risultati inerenti alle connessioni con Sysdig, lo strumento descritto nella sezione 2.1.
Sysdig nella sua forma base ha un comportamento molto simile ad eBPF, in quanto utilizza dei gestori nel kernel che consentono di monitorare gli eventi relativi a chiamate di sistema, che in questo caso specifico si occuperanno di monitorare gli eventi `connect, accept, sendto, recvfrom` relativi ai processi che utilizzano connessioni TCP e UDP per comunicare attraverso la rete. Al contrario di eBPF, Sysdig deve definire alcune regole supplementari per il filtraggio di questi eventi, in mancanza di tali regole lo strumento riporterebbe, ad esempio, anche informazioni inerenti a connessioni stabilite tra processi che comunicano all'interno del sistema operativo, che non sono di nostro interesse, come si può osservare nella figura 4.4 qui sotto.

```
root@riccardo-Lenovo:~# sysdig "evt.type in (connect,accept,sendto,recvfro
m)"
1102 09:22:05.425625454 1 nautilus (2662) > connect fd=27(<u>)
1103 09:22:05.425667219 1 nautilus (2662) < connect res=-111(ECONNREFUSED)
 tuple=0->ffff8ec4e3dd5000 /home/riccardo/.dropbox/command_socket
1601 09:22:05.453202361 0 nautilus (2486) > connect fd=27(<u>)
1602 09:22:05.453246851 0 nautilus (2486) < connect res=-111(ECONNREFUSED)
 tuple=0->ffff8ec4e980f800 /home/riccardo/.dropbox/iface_socket
6483 09:22:06.177220442 1 compiz (2307) > recvfrom fd=5(<u>) size=5856
6492 09:22:06.177235667 1 compiz (2307) < recvfrom res=5856 data=.........
.......P.......P...............................R.......R.............. tu
ple=NULL
6567 09:22:06.177549609 1 gnome-terminal- (16185) > recvfrom fd=4(<u>) siz
e=1764
6572 09:22:06.177555756 1 gnome-terminal- (16185) < recvfrom res=1764 data
=.....................................P...........R...........U...........Q
....... tuple=NULL
6766 09:22:06.177999535 1 compiz (2307) > recvfrom fd=5(<u>) size=1748
6770 09:22:06.178005192 1 compiz (2307) < recvfrom res=1748 data=.........
.............................P...........R...........U...........Q....... tu
ple=NULL
6976 09:22:06.178413771 0 syndaemon (2317) > recvfrom fd=3(<u>) size=4
6978 09:22:06.178416146 0 syndaemon (2317) < recvfrom res=-11(EAGAIN) data
= tuple=NULL
```

Figura 4.4: Esecuzione di sysdig in assenza di regole

Nello scenario seguente aggiungeremo le regole che ci servono per mettere a paragone lo strumento Sysdig con `NetPagefault`, in questo caso le regole che andranno aggiunte sono relative al filtraggio degli eventi processati da sysdig, ovvero, bisogna definire dei filtri in modo tale da poter ricevere solo le informazioni relative alle connessioni TCP e UDP.
L'input vero e proprio di sysdig sarà:

```
sysdig "evt.type in (connect,accept,sendto,recvfrom) and fd.type=ipv4
       and fd.l4proto in (tcp,udp)"
       -p"%proc.name) %evt.type %fd.l4proto:  tuple:[%fd.name]"
```

Dove le prime due righe servono per filtrare solo gli eventi `connect,accept,sendto,recvfrom` relativi ai protocolli di livello trasporto TCP e UDP, che utilizzano indirizzi IPv4, e la terza riga serve per filtrare l'output in questa forma:

```
nomeProcesso) tipoDiEvento, Protocollo:  [ip_src:port_src; ip_dest:port_dest]
```

Nella figura 4.5 viene mostrato il funzionamento di entrambi li strumenti.

```
^Croot@riccardo-Lenovo:~# sysdig "evt.type in (connect,accept,sendto,recvf
m) and fd.type=ipv4 and fd.l4proto in (tcp,udp)" -p"%proc.name) %evt.type
%fd.l4proto: tuple:[%fd.name]"
Plex) recvfrom udp: tuple:[0.0.0.0:1900]
Plex) recvfrom udp: tuple:[10.101.49.136:1901->10.101.61.105:1900]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:43988->10.101.49.136:19
01]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:41963->10.101.49.136:19
01]
Plex) recvfrom udp: tuple:[10.101.49.136:1901->10.101.61.105:1900]
Plex) recvfrom udp: tuple:[10.101.49.136:1901->10.101.61.105:1900]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:45974->10.101.49.136:19
01]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:56259->10.101.49.136:19
01]
Plex) recvfrom udp: tuple:[10.101.49.136:1901->10.101.61.105:1900]
Plex) recvfrom udp: tuple:[10.101.63.96:39695->10.101.61.105:1900]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:49777->10.101.63.96:396
95]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:48708->10.101.63.96:396
95]
dnsmasq) sendto udp: tuple:[0.0.0.0:42819]
dnsmasq) sendto udp: tuple:[10.101.61.105:42819->131.114.33.20:53]
dnsmasq) sendto udp: tuple:[10.101.61.105:42819->131.114.33.20:53]
dnsmasq) sendto udp: tuple:[10.101.61.105:42819->131.114.33.21:53]
dnsmasq) recvfrom udp: tuple:[10.101.61.105:42819->131.114.33.21:53]
dnsmasq) recvfrom udp: tuple:[10.101.61.105:42819->131.114.33.20:53]
dnsmasq) recvfrom udp: tuple:[10.101.61.105:42819->131.114.33.20:53]
dnsmasq) recvfrom udp: tuple:[10.101.61.105:42819->131.114.33.21:53]
Plex) recvfrom udp: tuple:[10.101.63.96:39695->10.101.61.105:1900]
Plex) recvfrom udp: tuple:[10.101.61.232:50877->10.101.61.105:1900]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:45584->10.101.61.232:50
877]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:41380->10.101.61.232:50
877]
Plex) recvfrom udp: tuple:[10.101.61.232:50877->10.101.61.105:1900]
Plex) recvfrom udp: tuple:[10.101.55.94:59430->10.101.61.105:1900]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:35791->10.101.55.94:594
30]
Plex DLNA Serve) connect udp: tuple:[10.101.61.105:39877->10.101.55.94:594
```

(a) Sysdig

```
Running for 1000 seconds or hit Ctrl-C to end.
^C

Tcp client side (sock.connect) :

PID   IPVER PROC_NAME          IPSOURCE        IPDEST          DPORT
12445 4     Plex DLNA Serve    10.101.61.105   10.101.60.232   59524
12444 4     Plex DLNA Serve    10.101.61.105   10.101.26.117   2869
12442 4     Plex DLNA Serve    10.101.61.105   10.101.60.232   59524
12443 4     Plex DLNA Serve    10.101.61.105   10.101.1.92     49962
4029  4     Chrome_IOThread    10.101.61.105   216.58.205.163  443
12446 4     Plex DLNA Serve    10.101.61.105   10.101.1.92     49962

Tcp server side (sock.accept) :

PID   IPVER PROC_NAME          IPSOURCE        IPDEST          LPORT

Udp server side (sock.accept) :

PID   IPVER PROC_NAME          IPSOURCE        IPDEST          LPORT
12449 4     host               125.176.72.137  235.186.72.139  72
1421  4     Plex Media Serv    125.176.72.137  235.186.72.139  72
4029  4     Chrome_IOThread    125.176.72.137  235.186.72.139  72

Udp client side (sock.connect) :

PID   IPVER PROC_NAME          IPSOURCE        IPDEST          DPORT
12441 4     Plex DLNA Serve    10.101.61.105   10.101.61.105   59754
10382 4     TaskSchedulerFo    127.0.0.1       127.0.1.1       53
12440 4     Plex DLNA Serve    10.101.61.105   10.101.61.105   59754

Page fault:

PID      NAME              IPVER      MIN_FLT
4029     Chrome_IOThread   4          680
```

(b) `NetPagefault` tool

Figura 4.5: Confronto tra Sysdig e `NetPagefault`

I pacchetti effettivamente processati da Sysdig sono maggiori di quelli di `NetPagefault`, perché le regole che abbiamo definito in precedenza vengono applicate solo quando i pacchetti arrivano in spazio utente, di conseguenza possiamo affermare che l'efficienza dello strumento Sysdig è molto più bassa se paragonato con `NetPagefault`.
Paragoniamo adesso il tool sviluppato, in corrispondenza dei pagefault da esso rilevati, con il comando `top`. Come si può vedere dalla figura 4.6, i processi chrome e ssh causano all'incirca li stessi pagefault, anche se in questo caso il pid di chrome non corrisponde con quello rilevato da `top`.

```
  PID USER      PR  NI    VIRT    RES    SHR S %MEM inMin COMMAND
20175 riccardo  20   0  591004  41568  32000 S  1,2  3636 gedit
 2307 riccardo  20   0 1499504 112052  63696 R  3,2    88k compiz
 1189 root      20   0  757608  73720  49432 S  2,1  356k Xorg
23641 riccardo  20   0 1439920 332516  77376 S  9,6 3,52k chrome
16185 riccardo  20   0  673608  43732  30928 S  1,3  9319 gnome-terminal-
26979 riccardo  20   0   44920   5356   4700 S  0,2   330 ssh
```

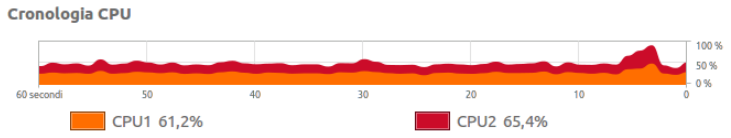
(a) `top` command

```
PID      NAME              IPVER      MIN_FLT
23156    Chrome_IOThread   4          3600
26979    ssh               4          425
```

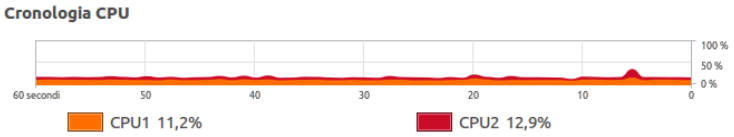
(b) `NetPagefault` tool

Figura 4.6: Confronto tra `top` e `NetPagefault`

Nella figura 4.7 vengono mostrati i grafici relativi al carico sulla CPU in presenza dello strumento Sysdig e in presenza di `NetPagefault`.



(a) Statistiche CPU Sysdig in esecuzione



(b) Statistiche CPU `NetPagefault` in esecuzione

Figura 4.7

In questo caso il carico sulla CPU rimane molto basso se paragonato a quello di Sysdig, anche in questo caso possiamo concludere che `NetPagefault` risulta essere più efficiente di Sysdig.

4.5 Lavoro futuro

Per via di alcuni limiti[2] presenti nel kernel da me utilizzato e per via del poco tempo avuto a disposizione per lo sviluppo di ulteriori funzionalità all'interno dei tool `TcpLatency` e `NetPagefault`, nella seguente sezione saranno presentati i lavori futuri correlati allo sviluppo di nuove funzionalità da aggiungere nei due tool da me sviluppati.
L'idea principale è quella di incorporare strumenti come `TcpLatency` con altri tool che monitorano le più comuni metriche di performance della rete, quali:

- **Throughput (Goodput)** per misurare la quantità di dati (nel caso del Goodput considerando solo il payload dei pacchetti) che possono essere mandati su un collegamento in uno specifico lasso di tempo;
- **Jitter** importante per le applicazioni multimediali;

Inoltre possono essere aggiunte funzionalità per monitorare la latenza applicativa, utilizzando i protocolli TCP/UDP.
Per quel che riguarda invece `NetPagefault`, vanno aggiunte le funzionalità per monitorare le comunicazioni dei processi che utilizzano indirizzi IPv6, si dovrebbe aggiungere inoltre la possibilità di monitorare le applicazioni che utilizzano la rete per ogni tipo di protocollo non solo di livello trasporto, come ARP, ICMP ed altri ancora. Infine si possono aggiungere ulteriori feedback provenienti dal sistema operativo, come i tempi di schedulazione dei processi che utilizzano la rete, o provenienti dall'hardware, come ad esempio le statistiche sull'uso della CPU di questi processi.

I codici dei due tool sono disponibili sul mio profilo github:
https://github.com/22RC/eBPF-TcpLatency-NetPagefault

[2]Mancano dei punti di attacco nel kernel come ad esempio: i tracepoint relativi al monitoraggio del protocollo UDPv6

Capitolo 5

Conclusioni

eBPF si è evoluto molto velocemente negli ultimi anni, sbloccando ciò che prima era fuori dal campo di applicazione del kernel. Ciò è stato reso possibile grazie all'incredibile facilità di analisi e progettazione, potente ed efficiente, fornita da eBPF. Attività che in precedenza richiedevano lo sviluppo di un kernel personalizzato ora possono essere ottenute, in modo più efficiente, con programmi eBPF, entro i limiti di sicurezza della sua macchina virtuale.

In questo elaborato sono stati presentati, in linee generali, i passi necessari per la creazione di strumenti di monitoraggio come `TcpLatency` e `NetPagefault`. Abbiamo analizzato le funzionalità e le tecniche adoperate per lo sviluppo di questi due tool che permettono di monitorare: la latenza di rete e i pagefault causati dai processi che comunicano tramite la rete.
I risultati ottenuti mostrano come sia possibile creare dei tool ad hoc e a basso overhead per monitorare il traffico di rete su sistemi Linux, permettendo quindi di analizzare e misurare le prestazioni delle applicazioni in scenari in cui molti altri tool non riescono ad osservare. Abbiamo visto che per eventi frequenti, come le attività di analisi del traffico, i costi generali di elaborazione possono diventare molto alti; eBPF, d'altra parte, può archiviare i dati nello spazio del kernel, e passare, solo le porzioni di risultati di nostro interesse, allo spazio utente.

Grazie ad eBPF si possono estrarre nuove metriche dal kernel e dalle applicazioni, ed esplorare l'esecuzione dei software in modo dettagliato a seconda dei nostri scopi.

Bibliografia

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[2] Andrew Begel, Steven McCanne, and Susan L. Graham. Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, page 123–134, New York, NY, USA, January 1999. ACM.

[3] Richard J HausmanLazar Birenbaum. Improved packet filtering for data networks. *Journal of Computer Science and Technology*, Jan 1997.

[4] Giovanni Cignoni, Carlo Montangero, and Laura Semini. verifica e validazione. 03 2018.

[5] Jonathan Corbet. Bpf: the universal in-kernel virtual machine https://lwn.net/articles/599755/. *News LWN*, (599755), 2014.

[6] Matt Fleming. A thorough introduction to ebpf https://lwn.net/articles/740157/. *News LWN*, (740157), 2017.

[7] Brendan Gregg. Linux performance analysis: New tools and old secrets (ftrace) https://www.slideshare.net/brendangregg/linux-performance-analysis-new-tools-and-old-secrets. 2014.

[8] DRAIOS INC. sysdig http://www.sysdig.org/. 2014.

[9] IOvisor. bcc https://github.com/iovisor/bcc. 2015.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 203–214, New York, NY, USA, 1998. ACM.

[11] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[12] J. Mogul. Efficient use of workstations for passive monitoring of local area networks. *SIGCOMM Comput. Commun. Rev.*, 20(4):253–263, August 1990.

[13] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 39–51, New York, NY, USA, 1987. ACM.

[14] Christian Seifert, Ramon Steenson, Ian Welch, Peter Komisarczuk, and Barbara Endicott-Popovsky. Capture – a behavioral analysis tool for applications and documents. *Digital Investigation*, 4:23 – 30, 2007.

[15] Suchakrapani Datt Sharma and Michel Dagenais. Enhanced userspace and in-kernel trace filtering for production systems. *J. Comput. Sci. Technol.*, 31(6):1161–1178, 2016.