Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Desenvolupament d'una eina basada en eBPF/XDP per al monitorage del rendiment d'una xarxa

---

Autor:
Gerard Vila Martínez

Tutors:
Jordi Paillisse Vilanova
Lluís Fàbrega Soler

MEMÒRIA

Convocatòria:
Juny 2023

Departament :
Arquitectura i Tecnologia de computadors

Projecte: Projecte Final de Grau
Document: Memòria
Títol: Desenvolupament d'una eina basada en eBPF/XDP per al monitorage del rendiment d'una xarxa
Autor: Gerard Vila Martínez
Data: Juny 2023

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor 1:
Jordi Paillisse Vilanova
Universidad Politécnica de Catalunya
Email: jordi.paillisse@upc.edu

Supervisor 2:
Lluís Fàbrega Soler
Universitat de Girona
Email: lluis.fabrega@udg.edu

Índex

Capítol 1

Introducció

1.1 Introducció

El monitoratge del rendiment d'una xarxa té cada vegada més importància en les xarxes actuals. L'obtenció de paràmetres estadístics que mesuren l'ocupació dels enllaços i el retard i pèrdues de paquets en un camí (retard mitjà, variació del retard, i percentatge de paquets perduts), permet conèixer l'estat de la xarxa i la detecció de mal funcionaments. A partir de l'anàlisi d'aquestes dades i de tècniques de predicció (basades en machine learning) els sistemes de gestió poden prendre decisions per aconseguir un ús eficient dels recursos de la xarxa.

Per altra banda, l'aparició de tecnologies com P4 (Programming Protocol-independent Packet Processors) o eBPF/XDP (extended Berkeley Packet Filter/eXpress Data Path) per a la programació de l'anomenat pla de dades (Programmable Data Plane o PDP), és a dir, per a la programació del comportament de dispositius de xarxa (commutadors, routers, tallafocs, NAT, etc.), permeten definir d'una manera més flexible com aquests dispositius processen i reenvien els paquets, i alhora obren la possibilitat de construir eines de monitoratge (i també en altres àmbits) més potents que les existents (observació de més variables, càlculs estadístics fets en el mateix dispositiu, processament més ràpid, etc.). En concret, eBPF és un conjunt d'instruccions i un entorn d'execució dins del kernel de Linux, que permet la programació del kernel en temps d'execució. eBPF, a més, es pot utilitzar per programar l'XDP, una capa de xarxa del kernel que processa els paquets ben a prop de la interfície de xarxa aconseguint, per tant, un processament més ràpid dels paquets.

1.2 Objectius

L'objectiu d'aquest projecte és desenvolupar una eina basada en eBPF/XDP per al monitoratge del rendiment d'una xarxa, concretament per obtenir paràmetres que mesurin el temps d'encuament i pèrdues de paquets en la cua de sortida d'un enllaç d'un dispositiu de xarxa, i també el retard i les pèrdues de paquets en un camí.

Per portar a terme aquest treball es seguiran els següents passos:

- Estudiar la tecnologia eBPF/XDP
- Estudiar els conceptes relatius al rendiment d'una xarxa i als paràmetres de mesura.

- Definir els requisits de l'eina de monitoratge.
- Construir l'eina de monitoratge.
- Construir les eines necessàries per a calcular els valors del rendiment.
- Fer proves per verificar el seu funcionament.

Capítol 2

Estudi de viabilitat

2.1 Viabilitat tècnica

Per a dur a terme el treball adequadament no és necessari comptar amb equipament molt específic. Principalment, és necessari que el sistema operatiu estigui basat en Linux 4.1 o superiors, ja que, es requereix una llibreria que necessita aquest requisit.

De cara als dispositius utilitzats durant el desenvolupament i testatge de l'aplicació, han sigut necessaris els següents recursos:

- Portatil
  - Windows 10
  - 8GB de memòria RAM
  - AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz
- Màquina virtual
  - Ubuntu 20.04
  - 2GB de memòria RAM

2.2 Viabilitat econòmica

El projecte s'ha implementat utilitzant programari amb llicències de codi lliure i el desplegament de l'aplicació s'ha dut a terme en un ordinador en local. Per tant, sense tenir en compte les hores invertides per part de l'estudiant, econòmicament aquest projecte no ha suposat cap inversió o despesa.

Tot i que el projecte no ha suposat cap despesa, s'ha fet una estimació d'un pressupost en el cas que s'hagués de contractar a un equip per a desenvolupar-lo. Un projecte com aquest es podria dur a terme amb un sol programador. S'estima que una versió definitiva es podria implementar en un mes a jornada completa. A la Fig. 2.1 es mostra el pressupost.

	€/hora	hores	€
Programador/a	14	70	980
Total		70	980

TAULA 2.1: Taula de pressupost

Capítol 3

Metodologia

La metodologia utilitzada és la metodologia **iterativa i incremental**. Aquesta, consisteix en desenvolupar l'aplicació en diferents cicles, on es fa un petit tros del projecte, anant incrementant a cada cicle el programa fins a obtenir l'objectiu final. Per tant, es poden entregar avenços abans de l'entrega final, permeten que es puguin canviar les coses tan bon punt s'entreguen.

Aquests cicles segueixen un seguit de passos per cada implementació que s'hagi de fer. Un cop acabat un cicle, es pensa i prepara el que es necessita pel següent cicle. Aquestes passes són les següents:

1. Establir els requisits a implementar
2. Analitzar els requisits
3. Fer codi i proves per aprendre i establir com fer els requisits
4. Desenvolupar els requisits
5. Comprovar els resultats i afegir comentaris sobre el resultat obtingut

El fet de fer una reunió cada setmana és el que ha fet que es faci servir aquesta metodologia.

Capítol 4

Planificació

La proposta es va presentar el 6 de febrer del 2023, amb data d'entrega el 2 de juny de 2023, tenint així aproximadament 4 mesos per a desenvolupar el projecte.

La planificació es va dividir en 4 blocs, cadascun d'un mes aproximadament, de la següent manera:

- Anàlisis
  - Estudiar i establir els requisits.
  - Analitzar la llibreria d'eBPF.
- Aprenentatge
  - Aprendre com funciona l'eBPF.
  - Fer proves, buscar exemples.
  - Testejar i comprovar el seu funcionament.
- Desenvolupament
  - Implementar una base simple, que pugui interactuar amb els paquets d'entrada.
  - Aplicar el mateix però a la sortida.
  - Interactuar amb els paquets.
  - Capturar informació del paquet i poder tractar-la.
  - Acabar d'agafar tota la informació necessària i fer comprovacions comparant-ho amb els valors de Wireshark.
- Documentació
  - Documentació d'estudis i decisions.
  - Redacció de la memòria.
  - Manual d'usuari.

La previsió temporal de cada tasca s'ha representat en un diagrama Gantt (vegeu la Fig. 4.1).

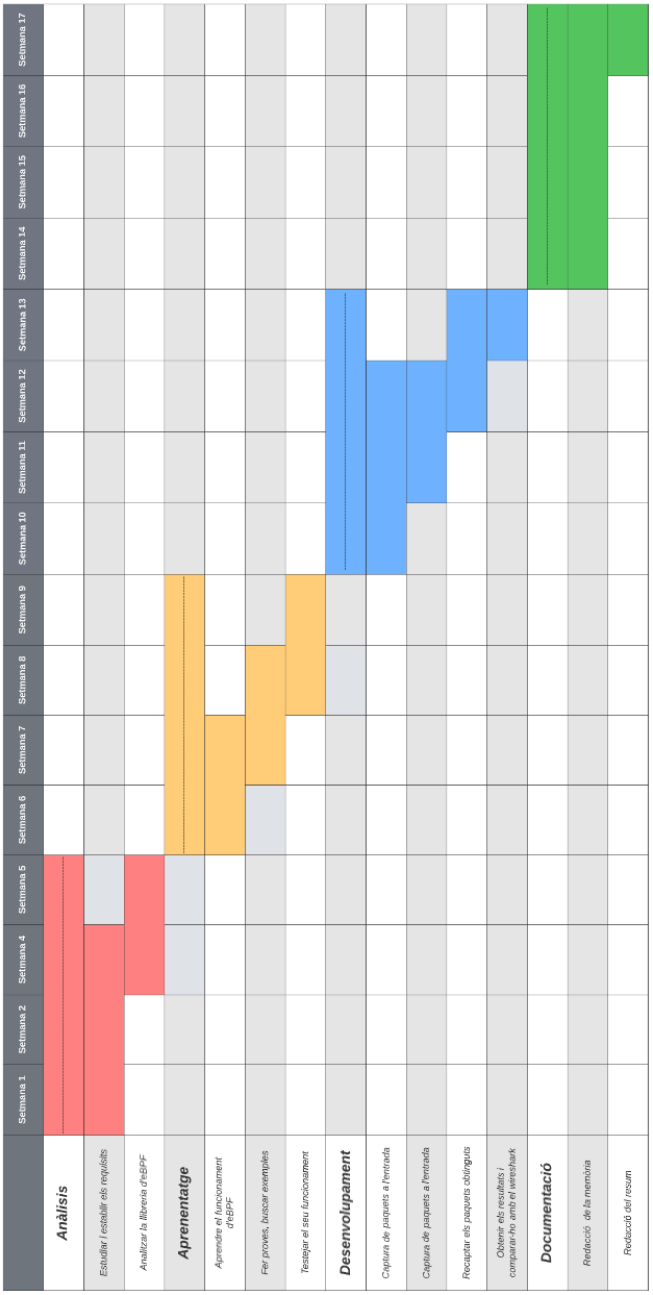


FIGURA 4.1: L'estructura del programa en l'escenari

Capítol 5

Marc de treball i conceptes previs

5.1 Paràmetres de rendiment d'una xarxa

El monitoratge del rendiment d'una xarxa té cada vegada més importància en les xarxes actuals. L'obtenció de paràmetres estadístics que mesuren l'ocupació dels enllaços i el retard i pèrdues de paquets en un camí (retard mitjà, variació del retard, i percentatge de paquets perduts), permet conèixer l'estat de la xarxa i la detecció de mal funcionaments.

5.1.1 Retard d'un paquet

En una xarxa de commutació de paquets un paquet és portat des d'una estació origen a una estació destí seguint un camí a través de la xarxa formada per una seqüència d'enllaços i nodes (commutadors). El retard d'un paquet és el temps que tarda a viatjar de l'estació origen a l'estació destí; més concretament, és el temps que passa des que s'envia el primer bit a l'entrada de la xarxa fins que arriba el darrer bit a la sortida de la xarxa.

Aquest retard és la suma dels temps de processament, els temps d'encuament, els temps de transmissió i els temps de propagació, en els nodes i enllaços del camí:

- Temps de processament en un node.

  Temps per decidir a través de quin enllaç de sortida el node ha de reenviar el paquet (llegir els camps del paquet, consultar taules de reenviament, etc.). Es tracta d'un temps que es pot considerar de valor fix.

- Temps d'encuament a la interfície de xarxa de l'enllaç de sortida del node.

  Com que hi ha diversos paquets que han de sortir a través de l'enllaç de sortida del node, els paquets s'emmagatzemen temporalment en una cua, fins que els toca ser enviats. Es tracta d'un temps variable, que depèn de la quantitat de paquets (o trànsit) a enviar, la velocitat de transmissió de l'enllaç de sortida (o capacitat de l'enllaç) i l'algoritme de la cua (FIFO, prioritat, round robin, etc.).

- Temps de transmissió del paquet a l'enllaç de sortida.

  És el temps que es tarda en transmetre els bits del paquet a l'enllaç, és a dir, n/rb, on n és la longitud del paquet en bits i rb la velocitat de transmissió de l'enllaç de sortida en *bps*.

- Temps de propagació de senyal a l'enllaç.

  És el temps que tarda el senyal que transporta el paquet en viatjar a través de l'enllaç que uneix un node amb el següent, és a dir, L/vp, on L és la longitud de l'enllaç en metres i vp és la velocitat de propagació de la llum en aquell enllaç m/s.

El retard d'un paquet es mesura amb paràmetres estadístics com el valor mig del retard, la variància del retard (o jitter) i també amb percentils.

En un únic node, la suma del temps de processament i del temps d'encuament d'un paquet es poden obtenir a partir de l'instant d'entrada del paquet al node i l'instant de sortida.

5.1.2 Pèrdua de paquet

Un paquet pot ser que es "perdi", és a dir, que no arribi a l'estació destí. Els paquets es poden perdre per diferents causes:

- Cues plenes

  Els paquets que surten a través de l'enllaç de sortida s'emmagatzemen temporalment en una cua, que té una determinada longitud. Si la cua està plena, un nou paquet que arriba no hi cap, i llavors es descarta (s'elimina).

  La cua s'emplena quan la velocitat de l'enllaç de sortida és menor que la velocitat d'entrada dels paquets a la cua. P.e., si entren 5 Mbps i surten 1 Mbps, la cua s'emplena més ràpidament del que es buida, i el temps d'encuament creix; si aquesta situació dura un cert temps, la cua s'emplena del tot (això depèn de la longitud de la cua) i llavors hi ha pèrdues de paquets.

- Errors i descarts

  El transport dels paquets no és perfecte, això fa que hi pugui haver error en aquests. Potser només és 1 bit el que ha sigut modificat, però aquest canvi fa que el missatge resultant no sigui l'inicial. Per això hi ha uns bits que serveixen per saber si el paquet és correcte o hi ha hagut algun canvi. Per tant, se sap que el paquet és erroni, aquest no es processa més i es descarta (s'elimina), i en conseqüència és un paquet perdut.

  També pot passar que el paquet que es rebi no tingui el format correcte; en aquests casos, el paquet també es descarta. Una altra situació en què el paquet es descarta és quan no se sap a través de quina interfície enviar-lo.

La pèrdua de paquets es mesura amb el paràmetre estadístic "% de pèrdua de paquets", definit com el nombre mig de paquets perduts dividit pel nombre de total de paquets enviats.

5.2 eBPF i XDP a Linux

eBPF és una tecnologia amb origen al kernel de linux que permet executar programes en un context privilegiat com el kernel del sistema operatiu(S.O.). Permet estendre la capacitat del kernel. A més, ho fa amb seguretat i eficiència i, encara més important, sense la necessitat de canviar el codi del kernel o carregar mòduls al kernel.

El S.O. garanteix seguretat i una execució eficient com si s'hagués compilat nativament amb l'ajuda d'un compilador i verificador JIT (Just-In-Time).

Avui en dia, eBPF s'utilitza per impulsar una gran varietat de casos d'ús:

- Proporcionar xarxes d'alt rendiment i balanceig de càrrega en centres de dades i entorns natius al núvol
- extreure dades d'observabilitat de seguretat detallades amb poca sobrecàrrega
- ajudar els desenvolupadors d'aplicacions a rastrejar aplicacions
- proporcionant informació per a la resolució de problemes de rendiment
- Aplicacions preventives i aplicacions de seguretat en temps d'execució

XDP (eXpress Data Path) proporciona una ruta de dades de xarxa programable d'alt rendiment al kernel de Linux. XDP proporciona processament de paquets al punt més baix de la pila de programari, just a l'instant en el que entren els paquets. Gran part de l'enorme guany de velocitat prové del processament de paquets RX (paquets entrants) directament des de la cua d'anell RX dels controladors, abans que es produeixi cap assignació d'estructures de metadades i, a més, està basada en eBPF.

Com es pot veure a la Fig. 5.1, al principi de tot hi ha el bloc XDP, la resta són tots els recorreguts que pot fer un paquet.

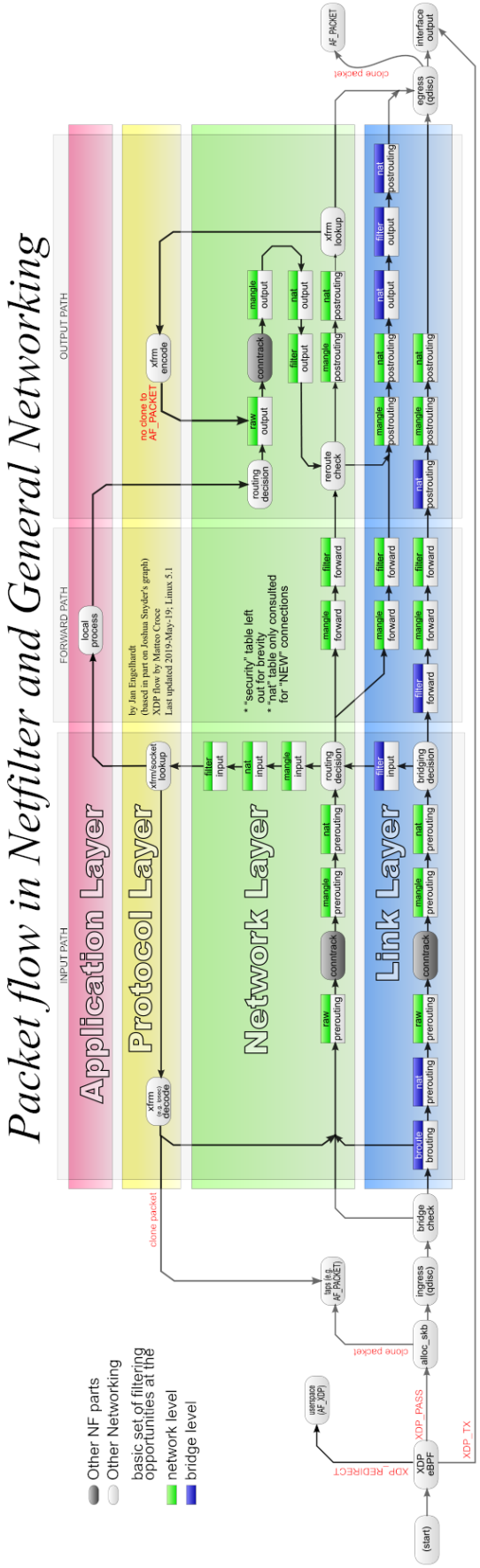


FIGURA 5.1: Packet flow paths in the Linux kernel [1]

5.3 Eina Bcc a python

BCC és un conjunt d'eines per crear programes eficients de control i manipulació del kernel, i inclou diverses eines i exemples útils. Fa ús de l'Extended BPF (Berkeley Packet Filters), conegut formalment com a eBPF, una nova característica que s'ha afegit per primera vegada a Linux 3.15. Gran part del que fa servir BCC requereix Linux 4.1 i superior.

BCC fa que els programes BPF siguin més fàcils d'escriure, amb C com a llenguatge en el kernel i Python i lua en el front-end. És útil per a moltes tasques, inclosa l'anàlisi del rendiment i el seguiment del trànsit de la xarxa.

Aquesta llibreria permet injectar el codi en el bloc d'XDP, que es troba just al principi de l'entrada dels paquets d'entrada. Això permet que el paquet es tracti tan aviat com es pugui, així tenim un rendiment més alt i obtenim dades més bones.

5.4 Traffic control a Linux

Tc s'utilitza per configurar el control de trànsit al kernel Linux.[2] El control de trànsit consisteix en el següent:

- FORMACIÓ (SHAPING)

  Quan al trànsit se li dona forma, la seva velocitat de transmissió està sota control. La formació pot ser més que reduir l'amplada de banda disponible, també s'usa per suavitzar les ràfegues de trànsit per a un millor comportament de la xarxa. Donar-l'hi forma es produeix a la sortida.

- PLANIFICACIÓ (SCHEDULING)

  Programant la transmissió dees paquets permet millorar la interactivitat del trànsit que ho necessita, encara que garanteix l'amplada de banda a les transferències massives. La reordenació també s'anomena priorització i només passa a la sortida. Principalment és el mètode que utilitza per escollir cada paquet en quin punt de la cua es posa.

- POLÍTICA (POLICING)

  Mentre que la formació tracta amb la transmissió del trànsit, la política es tracta amb el trànsit que arriba. La política així es produeix a l'entrada.

- CAIGUDES (DROPPING)

  El trànsit que superi una amplada de banda establerta es pot tirar immediatament, tant a l'entrada com a la sortida.

El processament del trànsit està controlat per tres tipus d'objectes:

- QDisc

  Qdisc és l'abreviatura de "queueing discipline"(disciplina d'encuament). Sempre que el kernel necessiti enviar un paquet a una interfície, es posa a la cua al qdisc configurat per a aquesta interfície.

- Classes

  Alguns qdiscs poden contenir classes, que contenen més qdiscs. El trànsit es pot posar a la cua a qualsevol dels qdiscs interiors, que estan dins de les classes.

- Filtres

  Un qdisc amb classe utilitza un filtre per determinar en quina classe es posarà en cua un paquet.

La llibreria iproute2, també fa us del TC, cosa que permet afegir, modificar o eliminar tant qdisc com classes com filtres.

Capítol 6

Requisits del sistema

En aquest apartat es descriuen els requisits necessaris del sistema, funcionals i no funcionals. Els requisits funcionals són els serveis que ofereix l'aplicació. D'altra banda, els requisits no funcionals són les necessitats del projecte a desenvolupar.

6.1 Requisits funcionals

Principalment, volem poder calcular el retard de tots els paquets que passin pel *router*. Per tant, és necessari poder obtenir el retard del paquet. Per obtenir el retard del paquet necessitem obtenir el moment d'entrada i el moment de sortida d'aquest. Per tant, hem de capturar el paquet en el moment d'entrar i en el de sortir, afegint una marca de temps en el moment que ha sigut interceptat. Doncs, es necessita estar el més proper a l'entrada i a la sortida per obtenir el valor més exacte. Així doncs, el lloc on s'ha de capturar aquests ha de ser el kernel.

Llavors, un cop s'ha capturat el paquet en el kernel, també és necessari poder extreure el valor a l'espai d'usuari per poder tractar-la.

- Accedir al punt més proper possible al bloc d'entrada del paquets al kernel
- Accedir el punt més proper possible al bloc de sortida dels paquets al kernel.
- Capturar els paquets entrants i sortints.
- Registrar l'instant d'arribada i sortida dels paquets.
- Calcular el temps del paquet al kernel.
- Extreure les dades del kernel a l'espai d'usuari.
- Guardar les dades aconseguides en un arxiu.
- Validar els resultats amb una eina externa.

6.2 Requisits no funcionals

- Un S.O. basat en Linux
- Com a mínim 2 interfícies de xarxa, amb una no es pot fer el reenviament.

Capítol 7

Estudis i decisions

7.1 Les màquines virtuals

Al principi, quan vam estar parlant sobre el projecte, teníem dubtes de si fer-ho en físic, utilitzant un router i ordinadors per crear una xarxa interna i provar-hi el programa, o utilitzar màquines virtuals, així simplificar l'escenari, ja que no hi hauria res físic, amb un ordinador es podria desenvolupar el projecte i provar de manera més senzilla.

El resultat va ser que vam decidir fer-ho amb les màquines virtuals, perquè simplificava el desenvolupament del projecte. Així es podia avançar sense necessitat d'una infraestructura gran.

També, com a opció a futur, vam dir de, un cop provat el programa en les màquines virtuals, mirar de fer l'escenari en físic i provar el codi.

7.2 El Sistema Operatiu de les màquines virtuals

El sistema operatiu que principal s'ha hagut de triar és el que tindrà tot el codi, ja que, aquest és el que necessitem que tot funcioni. Primer s'havia mirat l'Ubuntu 18.04, però, aquest, va donar problemes amb l'eina Bcc, així doncs, es va escollir l'Ubuntu 20.04.

Pel que fa a les estacions, aquestes no requereixen res especials. Per tant, d'entrada es van buscar màquines virtuals amb pocs requisits. Vam decidir agafar la màquina virtual que es va utilitzar a l'assignatura de Xarxes. Un Ubuntu 12.04, que té molt pocs requisits, permetent poder obrir una bona quantitat d'aquestes.

7.3 L'eina iperf per a la generació de trànsit

iPerf[3] és una eina per mesurar activament l'amplada de banda màxima possible a les xarxes IP. Admet l'ajustament de diversos paràmetres relacionats amb el temps, els *buffers* i els protocols (TCP, UDP, SCTP amb IPv4 i IPv6). Per a cada prova, informa de l'amplada de banda, la pèrdua i altres paràmetres.

Aquesta eina és molt útil per fer proves amb una gran quantitat de paquets de manera molt simple.

A més, és una eina molt útil per a fer una prova del codi creat, per veure el seu funcionament en casos amb moltes dades i molta velocitat.

7.4 L'eina Wireshark per a a l'anàlisi de trànsit

Wireshark és un analitzador de protocols utilitzat per a realitzar anàlisi i solucionar problemes en xarxes de comunicacions, tant per anàlisi de dades i protocols com també com a eina didàctica.

La funcionalitat que proveeix és similar a la de tcpdump, però afegeix una interfície gràfica i opcions d'organització i filtratge d'informació. Així, permet veure tot el trànsit que passa a través d'una o moltes xarxes a la vegada.

Wireshark està basada en la llibreria pcap.

La implementació de la llibreria pcap en Linux és coneguda com a Libpcap, mentre que a Windows es diu Winpcap.

La llibreria de captura de paquets Pcap proporciona una interfície d'alt nivell per als sistemes de captura de paquets. Tots els paquets de la xarxa, fins i tot els destinats a altres hosts, són accessibles mitjançant aquest mecanisme. També admet desar paquets capturats a un arxiu de guardat i llegir paquets des d'un arxiu guardat.

7.5 El format de dades JSON per emmagatzemar dades

Les dades s'han de poder guardar en un fitxer en ordre de no superar la memòria màxima que un programa té permès. Per això, l'estructura JSON és una estructura simple i molt útil per utilitzar. A més, existeixen moltes eines que ajuda a crear, llegir i modificar JSONs de manera simple. Per això, es va decidir usar aquest tipus d'arxiu per guardar les dades que anem obtenint.

A més, ens podem trobar amb el problema en llegir un arxiu JSON que aquest ocupa massa memòria i que el sistema operatiu mata el procés del programa. Per això, també hi existeixen eines, que s'han fet server en aquest projecte, que permeten llegir l'arxiu directament, així no cal carregar totes les dades de cop.

7.6 Editor LateX per a la redacció de la memòria

Latex és una eina que permet fer documents pdf de manera que no t'hagis de preocupar per l'estil, mentre se segueixi la sintaxi, el programa fa la resta. Afegit a això, en l'assignatura del projecte hi ha una plantilla, per tant, d'una forma molt simple s'ha pogut fer la documentació i sense preocupar-se per l'estil i la posició del text, imatges, índex, etc. .

Capítol 8

Anàlisi i disseny del sistema

8.1 L'escenari de treball

Per a desenvolupar el projecte, al principi, s'ha utilitzat el següent escenari(vegeu la fig. 8.1):

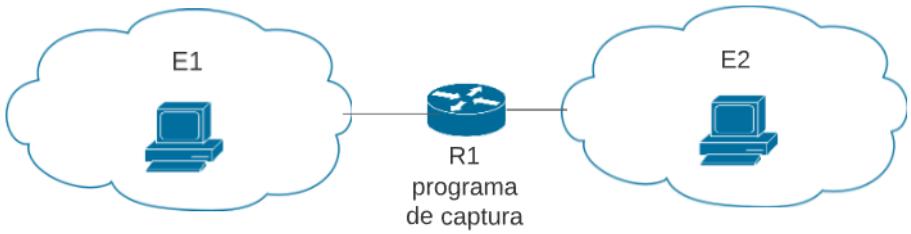


FIGURA 8.1: Escenari de treball 1

Com que la intenció del projecte és la d'obtenir el rendiment de la xarxa, el millor és fer la captura en el mig de tot. Per tant, l'estació que s'encarregarà de capturar els paquets farà el mateix que un router, reenviament de paquets (*ip forwarding*). Així doncs, com a un primer escenari, és requerit d'almenys dues estacions, així té sentit fer el reenviament, i d'un router.

En aquest escenari observarem el trànsit que es genera entre l'estació 1 i l'estació 2, utilitzant tant el *ping* com l'eina *iperf*. Així podrem veure el seu funcionament tant amb poc trànsit amb paquets *icmp* (*ping*) com amb molt trànsit amb paquets *tcp* (*iperf*).

Llavors, per complicar una mica més l'escenari, es va dissenyar aquest segon escenari (vegeu la fig. 8.2), així hi ha més estacions i trànsit.

En realitat, aquest segon escenari no deixa de ser una extensió de l'escenari anterior. Aquests permet tenir més trànsit transcorrent i en diferents enllaços. Així, una sola estació pot enviar paquets per més d'una estació, on passarà per diferents enllaços.

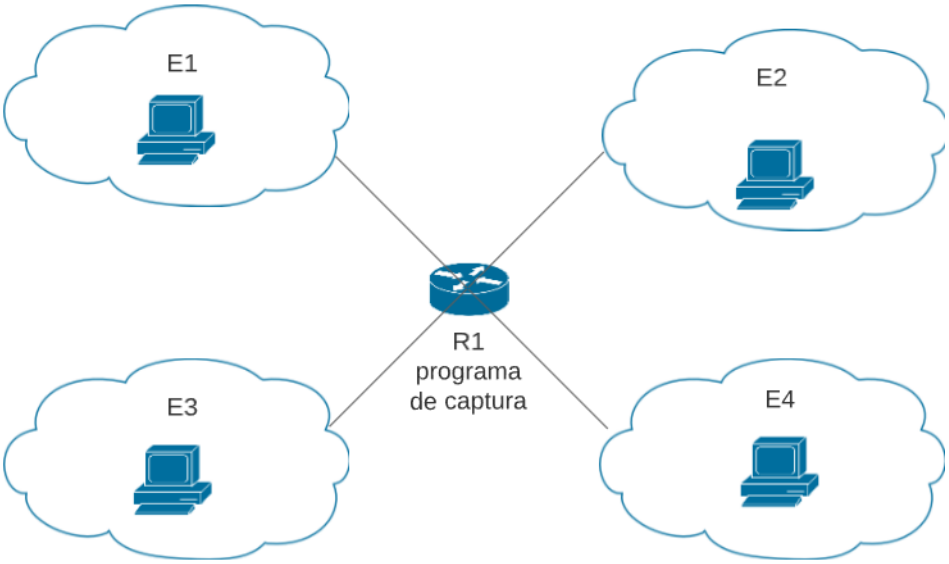


FIGURA 8.2: Escenari de treball 2

8.2 La identificació d'un paquet

Cada paquet conté una pila de protocols (p.e., HTTP dins TCP dins IP dins Ethernet), a cada protocol hi ha diferents camps amb el seu valor. Com que es necessita capturar el paquet en entrada i sortida, necessitem agafar uns camps que no es modifiquin en el procés de reenviament. En aquest cas, es va decidir que només els paquets amb el protocol IP es capturessin, ja que, aquests protocols, no modifica cap camp que ens és necessari en aquests procés i, a més, és el protocol que pot seguir al protocol ethernet. Observant els camps d'aquest protocol (veieu la Fig. 8.3), es va veure que cap d'aquests, per individual, permet identificar el paquet en entrada i sortida, però, un grup d'ells sí, sí que es pot identificar un paquet.

Aquests valors són els següents:

- L'adreça IP origen

  Sapigüent l'adreça IP, es pot filtrar de tots els paquets rebuts, aquells que corresponen a una sola estació. Però, en cas de rebre dos o més paquets a la vegada de la mateixa estació, no podem identificar quin és quin en entrar i sortir.

- L'identificador del paquet

  És un valor en hexadecimal que l'estació escull aleatòriament al principi d'una connexió. Comença amb le primer valor obtingut i hi va sumant 1 a aquest valor. Aquest valor juntament amb l'adreça IP de l'estació origen, es pot identificar els paquets quan entren i quan surten. L'únic problema que segueix és la fragmentació de paquets. Amb la fragmentació de paquets, el valor identificador és el mateix, per tant, dos fragments d'un paquet comparteixen el valor d'identificador, i, per descomptat, l'adreça origen.

- El número del fragment del paquet

  El *fragment offset*, el número de fragment, ens permet identificar i ordenar els diferents fragments d'un paquet.

Per tant, amb la suma dels tres valors podem identificar qualsevol paquet en l'entrada i en la sortida.

8.3 La captura d'un paquet entrant amb eBPF/XDP

Per a fer la captura dels paquets entrants, seguim més o menys el procediment de la pila de protocols. Es necessita anar desempilant els protocols que componen el paquet i decidir què fer amb aquests a cada protocol. En aquest cas, com que en arribar al protocol IP en tenim suficient, i aquest es troba just després del protocol ethernet, s'ha de desempilar un sol protocol.

Llavors, en entrar un paquet se seguirà els següents passos.

- En entrar un paquet, s'entra a la funció i el primer que es fa és guardar-nos una marca de temps, la que s'utilitzarà per calcular el retard.

- De l'estructura que rebem, obtenir les dades. Aquestes corresponen al paquet ethernet.

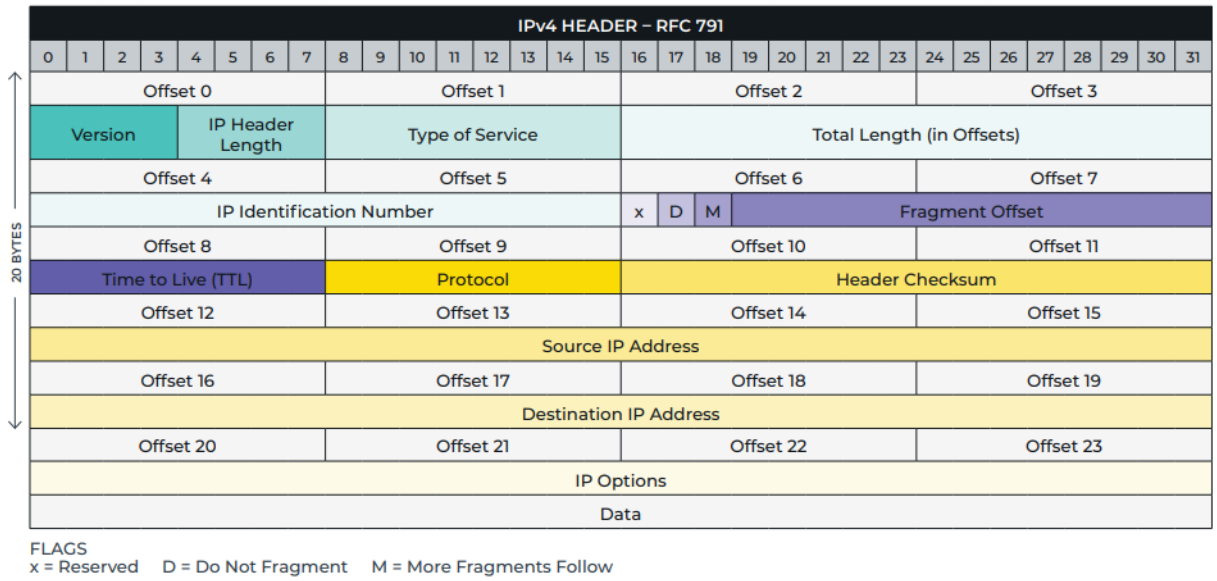


FIGURA 8.3: Capçalera dels paquets ipv4 [4]

- Desempilar el protocol ethernet del paquet, aconseguint així el paquet IP. En cas que no hi hagi el protocol IP es deixa passar el paquet i no ens guardem res d'ell.
- Obtenir la identificació del paquet 8.2.
- Guardar la informació necessària del paquet i la marca de temps al mapa de paquets. Aquests es guarda amb l'identificador del paquet com a clau.

Per tant, en arribar un paquet, la funció deixa passar aquells que no tenen el protocol IP i, dels que sí el tenen, ens guardem per a cada un d'ells una marca de temps d'entrada.

8.4 La captura d'un paquet sortint amb Traffic Control

Amb la sortida dels paquets s'ha de fer el mateix que en l'entrada, pel que fa a arribar al protocol IP.

Per tant, els primers passos que es requereixen són obtenir el paquet ethernet i descompilar el protocol ehternet per obtenir el protocol IP, en cas de no tenir aquest protocol, el deixem passar.

Un cop tenim el paquet IP, agafem la identificació del paquet. Aquest cop, com que estem a la sortida, s'ha de fer la comprovació de quan ha entrat el paquet. Per fer-ho, agafem l'estructura on guardem aquests paquets, el mapa de paquets, i busquem per la identificació del paquet (explicat a l'apartat 8.2).

Hi ha dues possibles opcions amb els paquets de sortida. Pot ser que es trobi l'identificador i, per tant, el paquet s'ha capturat en l'entrada i la sortida. En aquest cas, s'agafarà una altra marca de temps i s'actualitzarà el paquet en el mapa, així, amb les dues marques es pot calcular el retard. També s'afegirà un valor que representa que el paquet s'ha capturat en l'entrada i la sortida.

Iguament, pot passar que no es troba el paquet en el mapa. En aquest cas, que en principi no hauria de passar, vol dir que el paquet no ha entrat i només ha sortit. Això vol dir que aquest paquet l'ha enviat el router. S'ha tingut en compte, ja que s'ha utilitzat una estació com a router i, aquesta, pot enviar paquets com qualsevol altra estació. En el cas que això passes, aquest paquet es guardaria marcat com a paquet que no ha fet tot el recorregut, entrar i sortir. Per tant, serà contat com a paquet perdut.

8.5 Estructura del codi

8.5.1 La captura de paquets

Com hem comentat abans, l'eina Bcc permet compilar codi a eBPF i injectar-lo en el punt que necessites al kernel. Aquest eina està feta a python i compila codi escrit en C.

La funció que compila, requereix una cadena de text (*string*). Per tant, això permet tenir un sol arxiu de python, amb tot el codi a compilar a dins com a text. Es va decidir no fer-ho d'aquesta manera, escriure el codi a injectar, que s'escriu en C, en un arxiu a part i, a dins del codi en python llegir-lo i compilar-lo. Així tenir el codi separat i poder treballar-hi més còmodament.

Pel que fa al codi que captura els paquets, es va decidir tenir una funció pels paquets entrants i una pels sortints. Així, aquestes, són molt més simples i fàcils de comprendre, altrament podria ser bastant més complex de resoldre i entendre.

Per tant, el codi, en començar, obtindrà els codis de captura i l'injectarà al kernel, en el bloc que requereix cadascú.

8.5.2 L'estructura de guardar els paquets

Pel que fa a aquesta l'estructura on es guardarà la informació aconseguida, es va decidir utilitzar un mapa, concretament un *HASH_MAP* (vegeu la fig. 8.4). Com que els paquets, en principi, no segueixen cap ordre, el primer a entrar no està obligat a ser el primer a sortir, es necessita una estructura que tampoc requereixi cap ordre. Per tant, un mapa funciona en aquesta situació, on identifiquem el paquet amb la seva clau.

Doncs, es fa servir una mapa que com a clau fa servir l'identificador del paquet.

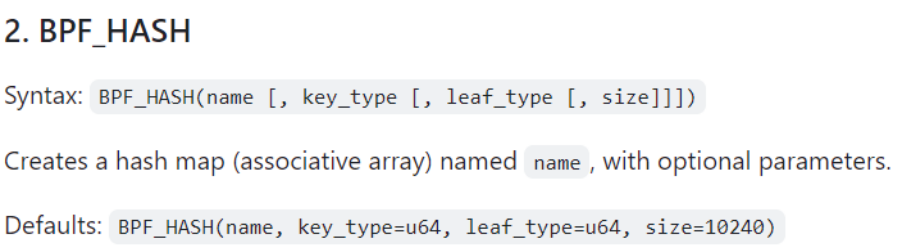


FIGURA 8.4: L'estructura per guardar els paquets capturats [5]

El punt important és que els dos codis que injectem poden fer ús d'aquesta mateixa estructura, encara que estiguin en punt diferent del kernel. Això permet tenir una certa unió entre els dos punts.

8.5.3 La recaptació dels paquets capturats

L'eina Bcc també dona cert accés al kernel. Permet obtenir accés a les estructures de dades que s'han compilat amb l'eina. Per tant, permet l'accés al mapa on guardarem els paquets capturats.

Aquest no ens avisarà de quan hi entren noves dades, en conseqüència, s'ha d'estar constantment observant l'estructura per comprovar si hi ha noves dades.

També permet la modificació de l'estructura. El principal que es fa servir en el projecte és el d'eliminar. S'eliminen els paquets que ja han sigut recaptats i, d'aquesta manera, buidem el mapa per no omplir-lo tot i que ens quedem sense espai on guardar més paquets. Per tant, mentre es guarden nous paquets es buiden els antics, per alliberar espai.

8.5.4 El guardat dels paquets

Durant el procés de recaptació es va emmagatzemant els paquets que ja han fet tot el recorregut, o que s'han perdut i ho sabem. S'emmagatzema en un arxiu JSON. Existeixen eines que guarden una estructura de dades en un arxiu JSON, però aquests requereix tenir totes les dades que es volen guardar, cosa que crea problemes amb la memòria. El resultat és que es van escrivint els paquets capturats a l'arxiu manualment, respectant l'estructura dels JSONs. Amb l'ús de la mateixa eina que permet guardar-ho tot, també permet convertir un objecte en una cadena, així obtenim les noves línies a escriure correctament.

8.5.5 L'aturada del programa

Un cop es vol parar el programa, es crea una interrupció de teclat (*ctr+c*). En fer-ho, para en sec la recaptació de paquets, però es continuen capturant més paquets, aquests no seran tractats. Tampoc guardarà els paquets que estiguin en espera de saber si són pèrdues o no.

Seguidament, es guarda la quantitat de pèrdues i acaba de completar l'arxiu JSON, així sigui un arxiu vàlid.

A continuació, ja no ens interessa continuar capturant paquets. Llavors, es recorre totes les interfícies on s'hi ha injectat els codis, tant a l'entrada (xdp) com a la sortida (qdisc).

Així doncs, ja no hi ha cap captura més i totes les dades han sigut guardares en un arxiu, que es farà servir per obtenir estadístics.

Per finalitzar, com es pot veure a la Fig. 8.5, tot depèn del programa principal. Aquest és el que posa tots els apartats, explicats anteriorment, a punt per fer la seva feina.

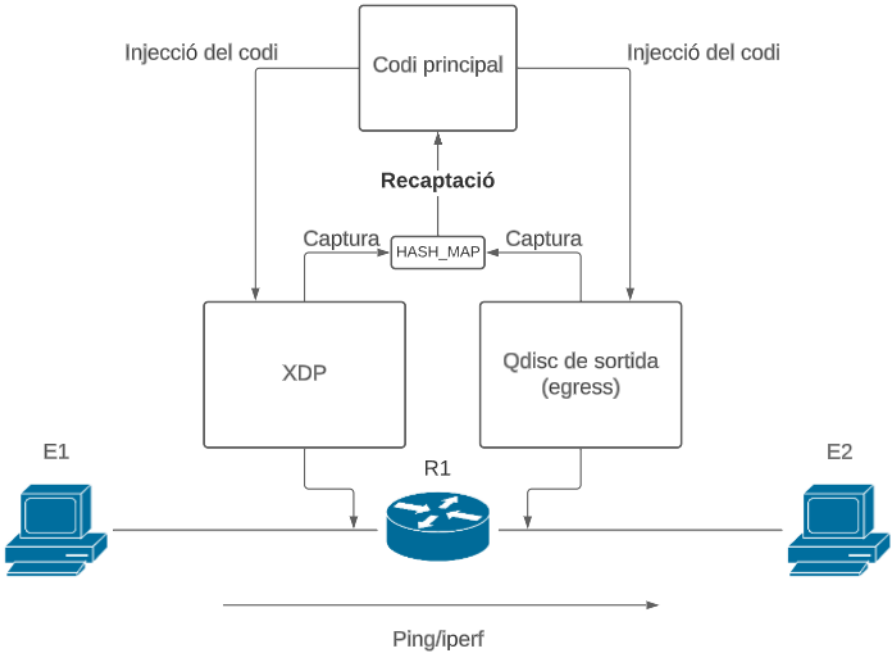


FIGURA 8.5: L'estructura del programa en l'escenari

Capítol 9

Implementació

En aquest punt, podem separar el codi en dos parts, la part que s'injectarà al kernel i el que es queda a l'espai d'usuari.

9.1 Captura de paquets entrants

Primer de tot, com que ja sabem on volem injectar la captura de paquets entrants, al bloc XDP, per tant, necessitem l'estructura de dades que utilitza aquest bloc. En el cas d'xdp l'estructura que utilitza és **xdp_md** (vegeu la Fig. 9.1).

```
struct xdp_md {
    __u32 data;
    __u32 data_end;
    __u32 data_meta;
    /* Below access go through struct xdp_rxq_info */
    __u32 ingress_ifindex; /* rxq->dev->ifindex */
    __u32 rx_queue_index;  /* rxq->queue_index  */

    __u32 egress_ifindex;  /* txq->dev->ifindex */
};
```

FIGURA 9.1: Estructura d'xdp al kernel [6]

Així doncs, la funció per a capturar els paquets rebrà aquesta estructura, com es pot veure a la Fig. 9.2.

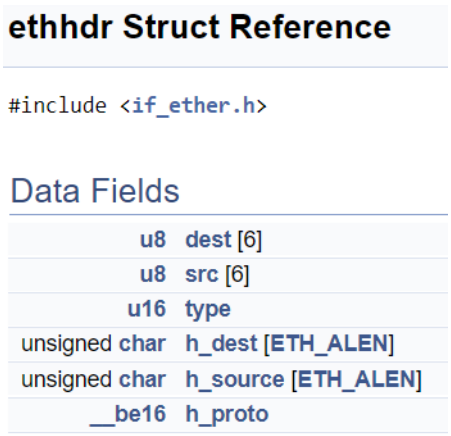
```
int ingress_capture(struct xdp_md *ctx){
```

FIGURA 9.2: Definició de la funció

Aquesta estructura és la referència a un paquet. Dins d'aquests hi trobem diferents estructures. La primera, que fa referència a la capçalera del paquet ethernet.

Aquesta estructura és eth_hdr (*ethernet header*), la capçalera d'un paquet ethernet (vegeu la Fig. 9.3).

Per tant, necessitem obtenir aquesta estructura. Per fer-ho, agafem les dades que conté el paquet, així ja haurem accedit al paquet ethernet.



ethhdr Struct Reference

`#include <if_ether.h>`

Data Fields

u8	dest [6]
u8	src [6]
u16	type
unsigned char	h_dest [ETH_ALEN]
unsigned char	h_source [ETH_ALEN]
__be16	h_proto

FIGURA 9.3: Packet flow paths in the Linux kernel [7]

Un cop hagim obtingut aquest valor, hem de comprovar que les dimensions del paquet són correctes (vegeu la Fig. 9.4). En cas que no estigui bé ho deixem passar, ja que, aquest codi no filtra, només hi és per obtenir informació.

```
struct ethhdr *eth = data;

nh_off = sizeof(*eth);
if (data + nh_off  > data_end)
    return XDP_PASS;
```

FIGURA 9.4: Obtenció del paquet ethernet

Igualment, encara no s'ha arribat on es necessita. Com que per identificar els paquets necessitem l'adreça IP origen, el camp identificador de la capçalera IP i el fragment *offset*. Tots aquests camps es troben en el paquet IP, per tant, necessitem anar a la següent capa del paquet.

Primer de tot ens hem d'assegurar que el paquet té aquesta capa, si no, no és necessari seguir, com es pot veure a la Fig. 9.5.

```
if(ntohs(eth->h_proto) != ETH_P_IP)
    return XDP_PASS;

struct iphdr *iph = (struct iphdr *)(data + nh_off);

nh_off += sizeof(struct iphdr);
if (data + nh_off > data_end)
    return XDP_PASS;
```

FIGURA 9.5: Obtenció del paquet IP

Un cop tenim la capçalera del paquet IP, només hem d'agafar les dades necessàries i guardar-les al mapa de paquets entrats (vegeu la Fig. 9.6). La clau d'aquesta entrada és un objecte amb els tres valors comentats abans, l'adreça IP origen, l'identificador i el número de fragment, en cas d'haver-se fragmentat. Llavors ja podem deixar que el paquet faci el recorregut que li toca.

Amb aquestes passes, ja s'ha guardat l'entrada d'un paquet amb una marca de temps.

```
struct packet_key new_key={};
new_key.src=src;
new_key.id=id;
new_key.n=frag;

packets.insert(&new_key,&new_packet);

return XDP_PASS;
```

FIGURA 9.6: Guardat d'un paquet

9.2 Paquets sortints

Com s'ha comentat anteriorment, els paquets de sortida les capturem al qdisc de sortida, per tant, no fa servir la mateixa estructura que s'utilitza a xdp.

En aquest cas es fa servir l'estructura **sk_buff** (vegeu la Fig. 9.7).

```
struct __sk_buff {
    __u32 len;
    __u32 pkt_type;
    __u32 mark;
    __u32 queue_mapping;
    __u32 protocol;
    __u32 vlan_present;
    __u32 vlan_tci;
    __u32 vlan_proto;
    __u32 priority;
    __u32 ingress_ifindex;
    __u32 ifindex;
    __u32 tc_index;
    __u32 cb[5];
    __u32 hash;
    __u32 tc_classid;
    __u32 data;
    __u32 data_end;
    __u32 napi_id;
    ...
};
```

FIGURA 9.7: Estructura d'skb al kernel [6]

Encara que és una estructura diferent, les dades continuen sent les mateixes, el que canvia és la forma amb la que es tracten. Ara fem servir un punter que el fem passar per les capçaleres per obtenir els valors que busquem. (vegeu la Fig. 9.8)

Per altra banda, l'obtenció del paquet en la sortida és igual a la d'entrada, únicament es diferencia que abans d'afegir les dades al mapa de paquets, primer mirem si aquest paquet ja ha entrat, així podrem obtenir els valors de la xarxa que busquem.

Per tant, continuem amb els mateixos passos. Obtenim el paquet ethernet.

Obtenim el paquet IP en cas que en tingui, si no, el deixem passar(el -1 fa referència que passa sense cap problema), com es pot veure a la Fig. 9.9.

Agafem els tres valors que fem servir per identificar el paquet (veure la Fig. 9.10).

```
int egress_capture(struct __sk_buff *ctx)
    u8 *cursor = 0;

    struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet)
    );

```

FIGURA 9.8: Obtenció del paquet ethernet

```
    if (ethernet->type != 0x0800) {
        return -1;
    }
    struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));

```

FIGURA 9.9: Obtenció del paquet ethernet

Un cop hem arribat al paquet IP i tenim les dades necessàries per identificar un paquet, s'ha de comprovar si el paquet que surt ha entrat, així afegint-hi una marca de temps per poder calcular el retard del paquet o, per altra banda, és un paquet que ha sortit, però que no ha entrat. En aquest últim cas, en principi, no hauria de passar, ja que, el que té aquest codi és un router i aquest no enviaria cap paquet, només hauria de reenviar paquets.

Per tant, en cas que el paquet no el trobem, guardem aquest paquet i, tanmateix, quedarà com a paquet perdut, perquè no entrarà cap paquet que coincideixi amb aquest. Però, en cas que el trobi, vol dir que el paquet ha entrat i ha sortit, per tant, hi afegim una marca de temps per poder calcular el retard d'aquest paquet, com es pot veure a la Fig. 9.11.

```
    struct packet_key new_key={};
    new_key.src=(int)ip->src;
    new_key.id=id;
    new_key.n=(int)ip->foffset;

```

FIGURA 9.10: Obtenció de la clau identificadora del paquet

```
struct packet updated_packet={};
    struct packet *old_pack=packets.lookup(&new_key);

    if(!old_pack){
        updated_packet.src_addr=(int)ip->src;
        updated_packet.dst_addr=(int)ip->dst;
        updated_packet.id=id;
        updated_packet.in_out=false;
        updated_packet.output_ns=(int)bpf_ktime_get_ns();


        packets.insert(&new_key,&updated_packet);
        return -1;
    }
    updated_packet=*old_pack;
    if(updated_packet.input_ns!=0)updated_packet.in_out=true;
    updated_packet.output_ns=(int)bpf_ktime_get_ns();


    packets.update(&new_key,&updated_packet);

    return -1;

```

FIGURA 9.11: Obtenció del paquet ethernet

9.3 Pèrdua de paquets

En capturar un paquet entrant no podem assegurar que aquest aconseguirà sortir, per tant, quan recollim les dades al codi principal, ens guardem l'identificador d'aquests, ja que encara no sabem si aquest sortirà. En guardar-lo, no ens podem permetre esperar tot el temps del món que aquests surti. Per tant, es guardarà al mapa de paquets perduts cada paquet que entri i una marca de temps, i s'eliminarà en el moment que es tingui constància que el paquet ha sortit correctament o, que hagi passat una certa quantitat de temps, on decidirem que el paquet s'ha perdut.

Els paquets entren i surten molt ràpid, per tant, s'ha decidit que amb un temps de 0.5 segons, determinat empíricament.

Per provar que això funciona, es necessita que hi hagi pèrdues. La manera més simple i fàcil ha sigut fer servir l'eina *iptables*. Aquesta eina ha permès crear un % de pèrdues en els paquets de reenviament.

9.4 Programa principal

Pel que fa al codi en python, que és el codi principal. Aquest és el codi que s'executa i compila i injecta els codis d'ebpf a xdp i al qdisc i que d'els paquets guardats al mapa, els extreu i es guarden en un fitxer JSON per poder tractar-los després.

9.4.1 La captura de paquets

Per tant, el primer que s'ha de fer és carregar l'arxiu que té les funcions a injectar. Donat el text que hi ha a l'arxiu, hi ha una funció de Bcc que compila el codi a eBPF a partir d'un text (vegeu la Fig. 9.12).

```
1 program = open('program.c', 'r').read()
2 b = BPF(text=program, cflags=["-w"])
3
```

FIGURA 9.12: Compilació a eBPF

Llavors, s'ha de carregar les dues funcions que s'han preparat en unes variables, per poder referenciar aquesta a l'hora d'injectar-ho (vegeu la Fig. 9.13).

```
1 ## INGRESS FUNCTION ##
2 funcio=b.load_func("ingress_capture", BPF.XDP)
3
4 ## EGRESS FUNCTION ##
5 egress_func=b.load_func("egress_capture",BPF.SCHED_CLS)
6
```

FIGURA 9.13: Obtenció de les funcions de captura de paquets

Ara queda injectar aquestes funcions a les seves respectives posicions. Pel que fa a la funció dels paquets d'entrada, es fa servir la funció *attach_xdp* (vegeu la Fig. 9.14).

Per altra banda, ara és on entra en joc l'eina del *TC(traffic control)* i *iproute2*. Ara ja no es serveix la funció *attach_xdp*, ja que ho estem injectant al qdisc i no a xdp. Amb la

```
1     b.attach_xdp(dev=inet, fn=funcio, flags=flags)
2
```

FIGURA 9.14: Injecció de la funció de captura de paquets entrants

llibreria iproute2 tenim una funció que ens permet injectar injectar aquesta funció al qdisc que es vulgui (vegeu la Fig. 9.15).

```
1     ipr.tc("add-filter", "bpf", idx, fd=egress_func.fd,name=egress_func.
      name, parent="0:", classid=1)
2
```

FIGURA 9.15: Injecció de la funció de captura de paquets sortints

9.4.2 La recaptació dels paquets capturats

Un cop injectat els codis de captura, només queda esperar. Recollir els paquets que es vagin guardant i posar-los en un arxiu (vegeu la Fig. 9.16). Per fer-ho, es fa servir un bucle infinit, fins que hi hagi una excepció de teclat. Així doncs, en tot moment va comprovant si hi ha entrat algun nou paquet.

En entrar un paquet, el primer a fer és comprovar si aquest paquet s'ha capturat tant a l'entrada com a la sortida. Això ho sabem mirant el valor de la variable *in_out* del paquet que hem guardat.

Aquest valor estarà a *true* només si ha entrat o sortit.

Per tant, un cop tenim aquest valor, sabem si el paquet ja el podem guardar al fitxer o si s'ha d'esperar a la sortida d'aquest paquet.

En el cas que ens hàgim d'esperar, ens guardarem l'identificador del paquet i, en el moment que el valor d'entrada i sortida estigui a cert, l'eliminarem d'aquest mapa de pèrdues, ja que, en el moment que surt no es considera com a pèrdua. Si no, en el moment que tenim la confirmació que ha fet el recorregut correctament, el podem eliminar del mapa de paquets capturats, però també s'ha de comprovar que estigui guardat al mapa de possibles pèrdues, en cas que hi sigui també s'eliminarà.

```
while True:
    try:
        for k,v in b["packets"].items():
            id_packet = {"src":ip2str(v.src_addr),"id":id2str(v.id),"
    frag_offset":v.n}

        if(v.in_out or ((json.dumps(id_packet) in losses) and timeout(
    losses[json.dumps(id_packet)],time.time()) ) ):
            if not v.in_out:
                lost+=1
            total+=1
            packet=''
            if n>1:
                packet=',\n'
            packet += json.dumps(packet2dict(v),indent=2,separators
    =(',',': '))
            f.writelines([packet])
            n+=1
            del b["packets"][k]
            if(json.dumps(id_packet) in losses):
                del losses[json.dumps(id_packet)]
            else:
            if(not json.dumps(id_packet) in losses):
                losses[json.dumps(id_packet)] = time.time()
                id_packet={}
    except KeyboardInterrupt:
        print("\nCaptured ",total," packets")
        print("Lost ",lost," packets")
        print("\nStopped package control")
        break

```

FIGURA 9.16: Recaptació del paquets capturats

9.4.3 El guardat dels paquets

Com s'ha comentat anteriorment, els paquets es guardaran en un arxiu JSON. Es podria utilitzar la funció *dump*. Aquesta funció, donat un diccionari, guarda tots els valors del diccionari en un arxiu, amb el format que toca. El problema amb aquesta funció és la memòria, es necessita tenir tota la informació en una variable per poder guardar-ho. Per tant, la solució és entrar en un arxiu manualment cada paquet. Per fer-ho s'utilitza una altra funció del JSON, la funció es diu *dumps*. Aquesta converteix un diccionari en una cadena que compleix el format dels arxius JSON. Així doncs, aquesta cadena pot ser escrita directament al fitxer. En aquest cas, la cadena és de més d'una línia. per tant, fem servir la funció *writelines* per escriure-les totes de cop (vegeu la Fig. 9.17).

```
packet+=json.dumps(packet2dict(v),indent=2,separators=(',',': '))
f.writelines([packet])

```

FIGURA 9.17: Recaptació del paquets capturats

9.4.4 L'aturada del programa

Durant tota la recaptació de paquets s'espera una interrupció de teclat. Aquesta interrupció aturarà la recaptació de paquets i començarà a desmuntar tot. En el moment que hi ha la interrupció de teclat, es para de guardar nous paquets i, primerament, es guarda el nombre de paquets perduts i s'acaba d'escriure a l'arxiu el que necessita per acabar de completar-se i ser vàlid.

Tan bon punt l'arxiu és complet, el tanquem, ja és necessari, i ens posem a treure el codi de captura del kernel. Per fer-ho s'ha d'utilitzar les mateixes eines amb les quals s'han injectat. Per la captura de paquets entrants a xdp s'utilitza l'eina Bcc. Per la captura de paquets sortints al qdisc s'utilitza l'eina TC. Això es fa per cada interfície en la qual s'ha injectat.(vegeiu la Fig. 9.18)

```
for inet in interface :
    idx = ipr.link_lookup(ifname=inet)[0]
    ## DELETE INGRESS ##
    b.remove_xdp(inet)
    ## DELETE EGRESS ##
    os.system("sudo tc filter del dev {}".format(inet))

```

FIGURA 9.18: Recaptació del paquets capturats

Capítol 10

Resultats

10.1 Captura de paquets sense pèrdues

Per engegar el programa només requereix executar el programa de python. Aquest requereix privilegis de superusuari.

Un cop es vol parar de capturar els paquets, simplement es fa una interrupció de teclat (*Ctr c*).

Llavors obtenim ja la quantitat de paquets que s'han perdut. Tot això es pot veure a la figura 10.1.

```
user@host:~/eBPF$ sudo python3 program.py
Start package control
^C
Captured  11777  packets
Lost  0  packets

Stopped package control
```

FIGURA 10.1: Execució del programa principal

Com podem veure, s'han capturat molts paquets, ja que, s'ha injectat trànsit amb l'eina *iperf* amb paquets *TCP*.

Aquest trànsit s'ha injectat entre dues estacions, una com a servidor (Figura 10.2) i el client (Figura 10.3), durant 2 segons.

```
01.06.23-xarxes@UbuntuX1:~$ iperf -s -V
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local ::ffff:2.2.2.4 port 5001 connected with ::ffff:1.1.1.4 port 51870
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 2.2 sec  14.2 MBytes  55.4 Mbits/sec
```

FIGURA 10.2: Execució de l'eina *iperf* a l'estació servidor

```
01.06.23-xarxes@UbuntuX1:~$ iperf -c 2.2.2.4 -t 2
------------------------------------------------------------
Client connecting to 2.2.2.4, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 1.1.1.4 port 51870 connected with 2.2.2.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.1 sec  14.2 MBytes  57.9 Mbits/sec
```

FIGURA 10.3: Execució de l'eina *iperf* a l'estació client

Això ha generat molts paquets, els podem trobar en un arxiu JSON, només els paquets capturats, encara que, en aquests casos, són tots. Amb aquest arxiu, utilitzant un petit programa, es mostra un histograma i calculem el retard mitjà dels paquets, el màxim i el mínim.

Llavors, com es veu a la fig. 10.1, s'han capturat 11777 paquets i no hi ha hagut cap pèrdua. Aquests paquets són tots els paquets que han anat d'una estació *A* a l'estació *B* i els de *B* a *A*. Per tant, per saber la quantitat que ha enviat A, si no hi ha cap pèrdua, serà la meitat.

S'ha posat una petita condició en l'histograma. Els temps més grans a 500 microsegons no es tindran en compte en l'histograma. Això s'ha decidit després de comprovar que si el retard màxim obtingut és massa gran, llavors no es pot veure bé la distribució dels temps resultants.

Si observem els paquets capturats, veurem que la gran majoria tarden menys de mil microsegons (vegeu la fig. 10.4).

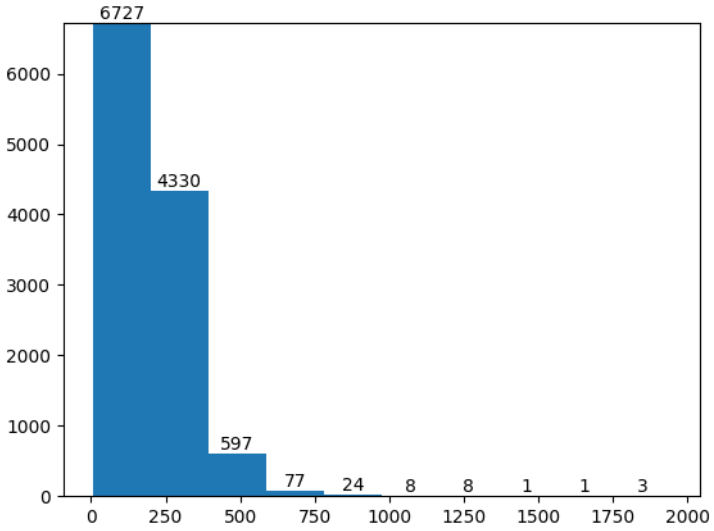


FIGURA 10.4: L'histograma del retard dels paquets capturats amb el programa

S'ha obtingut un retard mitjà de 187 microsegons (vegeu la fig. 10.5).

Si comparem el valor mitjà el retard més baix i el més alt, veiem que aquest és més proper al retard més baix. Això ens diu que la majoria dels paquets que arribin tardaran entre 0 i 187 microsegons.

```
user@host:~/eBPF$ python3 calculate.py output2.json
Màxim i mínim
1945.5710000000001 µs i 4.837 µs
Mitjana de retard
187.47311701766264  µs
```

FIGURA 10.5: L'execució del programa de processar els paquets del programa

Arribat aquí, no sabem del tot com són de correctes els valors obtinguts. En aquest cas, per poder fer una comparació de valor per veure com poden ser de vàlids

aquests valors, s'ha usat l'eina wireshark. Aquesta també agafa els paquets i ens permet importar-los en format JSON.

Aquest arxiu importat s'ha de reduir, hi ha moltes dades que no són necessàries, per tant, abans puguem comparar res, s'ha d'extreure el més necessari. Per fer-ho, utilitzem un petit programa, anomenat *comopute_json.py*, que guardar en un nou arxiu.

Un cop tenim les dades processades, ja es pot fer els mateixos càlculs que l'arxiu sortit del programa.

Així doncs, obtenim un histograma (veure fig. 10.6) semblant a l'histograma del resultant del programa(veure fig. 10.4).

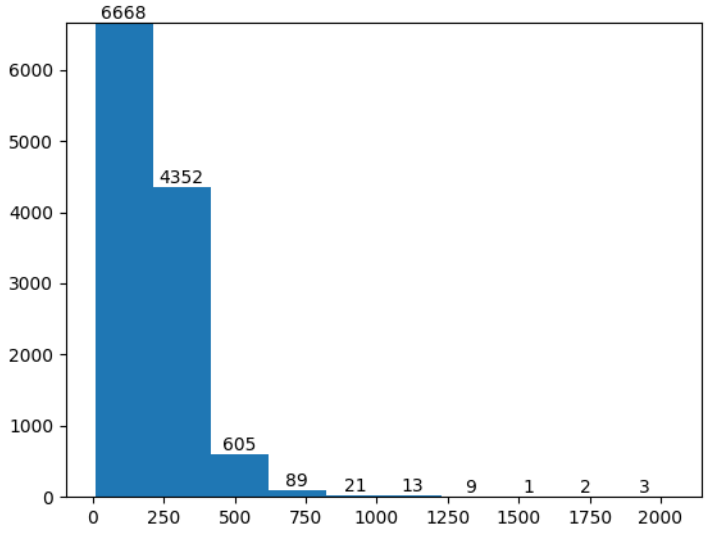


FIGURA 10.6: L'histograma del retard dels paquets capturats amb wireshark

Amb el wireshark s'ha obtingut un mitjana de retard de 200 microsegons (vegeu la fig. 10.7).

```
user@host:~/eBPF$ python3 calculate.py wireshark.json
Màxim i minim
2043.2269999999253 µs i 7.309999999996486 µs
Mitjana de retard
200.61323454900867  µs
```

FIGURA 10.7: L'execució del programa de processar els paquets del wireshark

Si comparem els resultats del programa amb les del wireshark,es pot veure que els dos histogrames tenen una forma idèntica en la distribució.

La forma resultant en la qual es mostren els valors és igual (vegeu la fig. 10.8). De fet, observant l'histograma que conté la diferència dels retards, es pot veure com quasi tots els valors estan entre 0 i 100 microsegons. Per tant, podem assumir que la diferència dels retards entre el wireshark i el programa, en la majoria dels casos, està entre 0 i 100 microsegons.

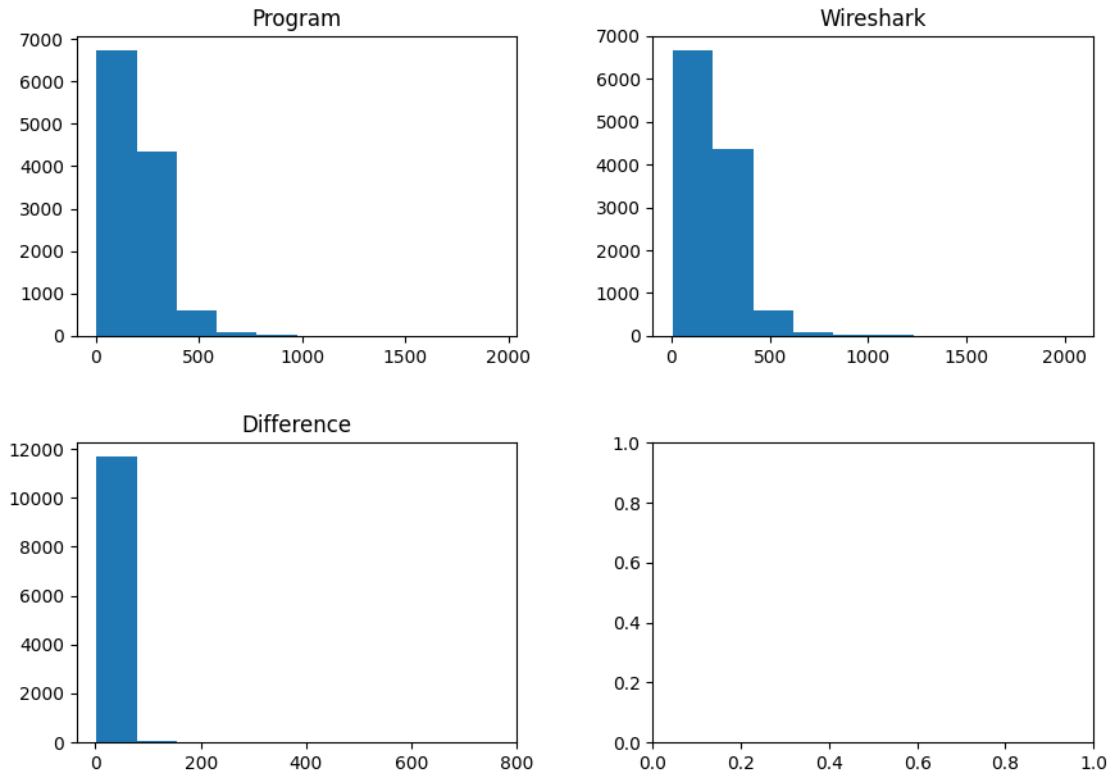


FIGURA 10.8: Comparació del programa i el wireshark amb un histograma

Si comprovem els valors de retard mitjà i de mínims i màxims, veiem que tant el mínim com el màxim són valors molt semblants, encara que una mica més grans pel cas del wireshark. Això correspon també en una mitjana més gran que la del programa. Encara que la diferència segueix bastant baixa (vegeu al fig. 10.9). La diferència de mitjana acaba sent de 13 microsegons. Per tant, la diferència de temps, aproximadament, de cadascú dels paquets capturats amb programa i el wireshark és de 13 microsegons.

```
user@host:~/eBPF$ python3 calculate.py output2.json wireshark.json
Mitjana WireShark
187.47311701766264  µs
Mitjana Programa
200.61323454900867  µs
Diferencia de mitjana
13.140117531346021  µs
```

FIGURA 10.9: L'execució del programa de processar els paquets comparant el programa amb wireshark

Com la diferència de retard dels paquets entre el programa i el wireshark, la majoria, tenen una diferència semblant i bastant baixa, es pot considerar que aquesta diferència és la diferència de temps entre que el programa agafa el paquet i el wireshark l'agafa.

Per tant, el fet d'obtenir valors molt semblants, podem considerar que els valors aconseguits amb el programa són vàlids.

10.2 Captura de paquets amb pèrdues

Pel que fa al cas de les pèrdues, depenent de la raó que hi hagi les pèrdues, el retard serà bastant gran o no. En aquest cas, s'ha utilitzat la comanda d'*iptables* que permet crear una probabilitat de perdre un paquet.

Per tant, no s'emplenarà la cua i el retard no serà gaire gran. Així doncs, aquest resultat és bastant simple, ja que, el punt d'aquests és el de veure que la captura de pèrdues és correcte.

Com es pot veure a la fig. 10.11, s'ha fet una transmissió de 5 paquets, dels quals 3 han sigut perduts. Si ho comprovem amb el resultat obtingut al programa (vegeu la fig. 10.10), veiem que hi ha el mateix nombre de pèrdues. També es pot veure que ha capturat 7 paquets, mentre que l'estació que ha fet el *ping* ha enviat 5 paquets.

Això és degut al fet que per cada paquet que arriba a l'estació destí, s'enviarà un paquet de resposta. Per tant, per cada paquet enviat, si no es perd, contarà com a dos paquets capturats, el paquet d'anada i el de tornada. Així doncs, en haver capturat 7 paquets i haver-ne perdut 3, vol dir que s'han capturat quatre paquets que, seguint el que s'ha dit anteriorment, són els dos paquets que no s'han perdut.

```
user@host:~/eBPF$ sudo python3 program.py
Start package control
^C
Captured  7  packets
Lost  3  packets

Stopped package control
```

FIGURA 10.10: Resultat de la captura de paquets

```
01.06.23-xarxes@UbuntuX1:~$ ping 2.2.2.4 -c5
PING 2.2.2.4 (2.2.2.4) 56(84) bytes of data.
64 bytes from 2.2.2.4: icmp_req=3 ttl=63 time=2.82 ms
64 bytes from 2.2.2.4: icmp_req=4 ttl=63 time=2.21 ms

--- 2.2.2.4 ping statistics ---
5 packets transmitted, 2 received, 60% packet loss, time 4004ms
rtt min/avg/max/mdev = 2.210/2.518/2.826/0.308 ms
```

FIGURA 10.11: Consola de l'estació que fa *ping*

També podem observar l'histograma d'aquests paquets capturats, però no té molta lògica, ja que no hi ha prou paquets per a extreure informació interessant. A més, els paquets perduts no es tenen en compte, ja que, en perdre's no s'ha pogut capturar cap marca de temps en la sortida.

Així doncs, obtenim un histograma amb tots els valors molt propers (vegeu la fig. 10.13).

I el valor mitjà no és gaire realista, comparant amb el resultat anterior, ja que no hi ha prous valors per agafar-ho com a vàlid (vegeu la Fig. 10.12).

```
user@host:~/eBPF$ python3 calculate.py output2.json
Màxim i mínim
48.978 µs i 11.326 µs
Mitjana de retard
29.793999999999997  µs
```

FIGURA 10.12: Consola del valors obtinguts

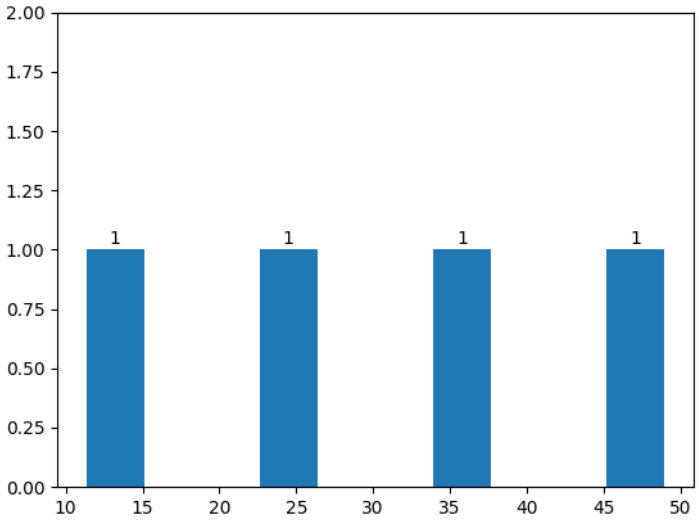


FIGURA 10.13: Histograma de la captura de paquets amb pèrdues

Capítol 11

Conclusions

L'objectiu del projecte era desenvolupar una eina basada en eBPF/XDP per al monitoratge del rendiment d'una xarxa, per obtenir paràmetres que mesurin el temps d'encuament i pèrdues de paquets en la cua de sortida d'un enllaç d'un dispositiu de xarxa.

Al final s'ha acabat amb els següents resultats:

- S'accedei al bloc xdp a l'entrada i al bloc del qdisc a la sortida dels paquets al kernel.
- Es fa la captura els paquets entrants i sortints i es recol·lecten en un arxiu.
- Es calcula el temps del paquet, la quantitat de paquets capturats i la quantitat de paquets perduts al kernel.

Per tant, podem considerar que s'han assolit els objectius. Però, també es podria considerar que el projecte no està finalitzat, ja que, a partir dels resultats aconseguits, es pot mirar d'obtenir altres valors de la xarxa per obtenir altres valors, i no només de dins del router sinó també dels enllaços que hi pot haver en una xarxa.

Bibliografia

[1] Jan Engelhardt. *Netfilter-packet-flow*. 2019. URL: https://en.wikipedia.org/wiki/File:Netfilter-packet-flow.svg.

[2] *tc(8) — Linux manual page*. URL: https://man7.org/linux/man-pages/man8/tc.8.html.

[3] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. URL: https://iperf.fr/.

[4] 2019. URL: https://www.gigamon.com/content/dam/resource-library/english/guide---cookbook/gu-bpf-reference-guide-gigamon-insight.pdf.

[5] Bcc. *bcc Reference Guide*. URL: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#maps.

[6] URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h.

[7] *ethhdr Struct Reference*. 2013. URL: https://docs.huihoo.com/doxygen/linux/kernel/3.7/structethhdr.html.