Alma Mater Studiorum - Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Dipartimento di Informatica - Scienze e Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Message-oriented Middleware per Routing Acceleration basato su eBPF/XDP in Contesto Serverless

Candidato
Feres M'hadhbi

Relatore
Chiar.mo Prof. Ing. Paolo Bellavista
Correlatori
Ing. Andrea Garbugli
Ing. Andrea Sabbioni

Anno Accademico 2021-2022
Sessione I

Ai miei genitori

Introduzione

Negli ultimi anni la necessità di processare e mantenere dati di qualsiasi natura è aumentata considerevolmente, in aggiunta a questo, l'obsolescenza del modello centralizzato ha contribuito alla sempre più frequente adozione del modello distribuito. Inevitabile dunque l'aumento di traffico che attraversa i nodi appartenenti alle infrastrutture, un traffico sempre più in aumento e che con l'avvento dell'*IoT*, dei *Big Data*, del *Cloud Computing*, del *Serverless Computing* etc., ha raggiunto picchi elevatissimi. Basti pensare che se prima i dati erano contenuti in loco, oggi non è assurdo pensare che l'archiviazione dei propri dati sia completamente affidata a terzi. Così come cresce, quindi, il traffico che attraversa i nodi facenti parte di un'infrastruttura, cresce la necessità che questo traffico sia filtrato e gestito dai nodi stessi.

L'obbiettivo di questa tesi è quello di estendere un Message-oriented Middleware, in grado di garantire diverse qualità di servizio per la consegna di messaggi, in modo da accelerarne la fase di routing verso i nodi destinazione. L'estensione consiste nell'aggiungere al Message-oriented Middleware, precedentemente implementato, la funzione di intercettare i pacchetti in arrivo (che nel caso del *middleware* in questione possono rappresentare la propagazione di eventi) e redirigerli verso un nuovo nodo in base ad alcuni parametri. Il *Message-oriented Middleware* oggetto di tesi sarà considerato il *message broker* di un modello *pub/sub*, pertanto la redirezione deve avvenire con tempi molto bassi di latenza e, a tal proposito, deve avvenire senza l'uscita dal *kernel space* del sistema operativo. Per questo motivo si è deciso di utilizzare eBPF, in particolare il modulo XDP, che permette di scrivere programmi che eseguono all'interno del kernel.

Per questa tesi è stata prima necessaria una fase di studio del preesistente *middleware* TEMPOS, in seguito, un'analisi di eBPF come tecnologia, più in dettaglio di XDP, dapprima teoricamente e poi è stato possibile creare piccoli programmi da inserire nel kernel. Questo passaggio è stato fondamentale in quanto ha permesso di capire la logica attraverso il quale il *Verifier* (modulo di eBPF che garantisce la sicurezza del codice scritto) permetta di iniettare il codice scritto nel kernel del sistema operativo. Successivamente dopo un veloce studio dei concetti base del linguaggio Rust, si sono analizzate le librerie che permettono di utilizzare questa tecnologia.

Sono state individuate e testate alcune librerie per poi convergere verso la libreria libbpf-rs, risultata più completa. Infine, per la fase di testing sono state utilizzate tre macchine appositamente preparate, si sono considerate diverse situazioni di carico del Message-oriented Middleware e paragonate le modalità di esecuzione del programma XDP. Alcune schede di rete, infatti, come quella utilizzata per la fase di test, permettono l'esecuzione di XDP in modalità *native*, ovvero permettendo di caricare il programma XDP direttamente sul driver della scheda di rete.

I capitoli di questa tesi sono stati organizzati in maniera tale da fornire un'idea sul contesto di lavoro, introducendo i concetti chiave relativi a vari ambiti di interesse per questa tesi. Inizialmente verranno introdotti i concetti relativi a *Internet of Things*, *Cloud Computing* e *Serverless*. In seguito, si descriverà il *middleware* TEMPOS sulla quale si è lavorato, descrivendone quali sono i componenti e il funzionamento, in modo da permettere di comprendere e discutere le scelte progettuali effettuate. Essendo un punto fondamentale per questa tesi ne verranno anche mostrate le prestazioni. Successivamente si discuterà in dettaglio la tecnologia eBPF con approfondimento di XDP per mostrarne le caratteristiche e il processo di utilizzo. Infine, verrà descritto il progetto con tutte le scelte progettuali seguito da un capitolo relativo alle prestazioni ottenute.

Indice

Elenco delle figure

Listings

Capitolo 1

Background

Contents

1.1 Internet of Things

Negli ultimi 30 anni si parla di "Network society" come conseguenza dello sviluppo tecnologico che ha assunto un ruolo importante nei cambiamenti dei principali meccanismi che contraddistinguono aspetti culturali, di comunicazione, politici ecc.. Le nuove tecnologie come i social media, il *cloud*, i *big data*, *IoT* e così via, hanno profondamente influenzato come la società interagisce e si trasforma. Per fornire una visione semplicistica, si può affermare che l'accento sull'interconnessione della società sia marcato principalmente dall'avvento dell'*Internet of Things*, che inevitabilmente introduce una grandissima mole di dati, con la necessità di essere processati per trarne valore, attraverso infrastrutture potenzialmente locate nel *cloud*. Tra i fattori abilitanti del propagarsi di questa nuova tecnologia c'è sicuramente il miglioramento tecnologico di sensori di varia natura, così come l'espansione della connettività e delle reti, infatti, sempre più dispositivi hanno la capacità di connettersi ad una rete o semplicemente ad altri dispositivi. Altri fattori abilitanti possono essere il miglioramento dei microprocessori, maggiore capacità di calcolo in minor spazio, potenza di calcolo più accessibile da chiunque e in qualsiasi momento, grazie

ai servizi forniti dai *cloud provider*, e infine forse quello con un impatto più grande, l'avvento dei social media, che ha inevitabilmente aumentato la mole di dati generati in maniera esponenziale, basti pensare che solo attraverso Facebook vengano generati 4 petabytes di dati al giorno. Quello che rappresenta la parola "Things" è una moltitudine di dispositivi di diversa natura, da dispositivi mobile a sensori industriali. Essi generano dati e li forniscono in input ad algoritmi che attraverso degli attuatori prendono decisioni su quale azione compiere. *Internet of Things*, oltre ad aver ampliato alcuni domini come quello dei *big data* [4], può assumere significati diversi, in contesti diversi, Tarkoma e Katasonov affermano che l'IoT sia "una rete globale e un'infrastruttura di servizio con una densità e connettività variabili e con una capacità di auto-configurazione basata su formati e protocolli stardard ed interoperabili, consistono in entità che hanno un'identità, attributi fisici e virtuali e sono perfettamente interconnessi in modo sicuro ad Internet." [11]
L'avvento dei *big data* ha reso obsoleti i classici sistemi di calcolo e le infrastrutture di comunicazione, c'è bisogno quindi di nuovi paradigmi e ad esempio, in base alla località dell'esecuzione o dello storage si può differenziare tra *cloud* e *edge computing*. Mentre questo argomento verrà trattato meglio nella Sezione 1.2 , soffermandosi su come IoT possa cambiare le nostre vite si può vedere come abbia introdotto innumerevoli benefici in diversi settori, tra cui:

- Healthcare
- Education
- Smarthome
- Smartcities
- Industria

Nell'ambito dell'*healthcare*, per esempio, IoT ha introdotto nuove possibilità sul campo del monitoraggio di un paziente con invio di dati ad alcuni dispositivi direttamente consultabili dal medico curante o dal paziente stesso. Un altro importante pilastro è rappresentato dai *wearable devices*, un esempio potrebbe essere un dispositivo in grado di iniettare insulina nei pazienti diabetici sulla base di dati monitorati. La sfida per questa malattia consiste nel tenere l'insulina nel corpo ad un livello costante, se troppa, si ha un calo di zuccheri, se troppo poca si rischia il danneggiamento di organi interni o di contrarre altre malattie autoimmuni. IoT è utile in questa casistica in quanto, considerando due componenti interconnessi, una pompa di insulina e un sensore di monitoraggio del glucosio nel sangue, questi comunicano e attraverso un algoritmo che tiene in considerazione diversi aspetti, tra cui l'intervallo di tempo tra la misurazione e l'invio dei dati, riesce a iniettare il giusto quantitativo di insulina nel paziente mantenendo il livello di glucosio nel sangue costante.

Un altro campo altrettanto importante è quello delle *smart cities*, in particolare, tra i problemi risolti a livello urbano attraverso l'utilizzo di sensori e *IoT* si può trovare il bilanciamento del traffico stradale, la gestione semafori in base al traffico, la gestione e distribuzione dell'energia elettrica, in generale, tutto ciò che in qualche modo può migliorare lo stile di via dei cittadini con un occhio anche al rispetto per l'ambiente. Diverse, tuttavia, sono le sfide da tenere in considerazione, sempre più importante, la questione della privacy, la sicurezza, l'affidabilità, la scalabilità ecc.. Sfide che riguardano anche l'aspetto tecnologico, infatti spesso *IoT* per le caratteristiche elencate utilizza, ad esempio, tecnologie *big data* per mantenere e processare i dati (nonostante *IoT* sia più concentrata sui flussi di dati piuttosto che su grandi quantità mantenuti in memoria), il *cloud* per garantire maggiore scalabilità, *machine learning* per il riconoscimento di pattern, e così via.

1.1.1 Architettura e componenti

Come si può vedere in Figura 1.1, in un sistema *IoT* si possono trovare diversi componenti che interagiscono tra di loro. Tra questi abbiamo:

- Sensor
- Communication channel
- External utilities
- Decision trigger

I *sensor* sono dispositivi fisici in grado di fornire dati basati su misurazioni effettuate nell'ambiente di competenza, nell'esempio di prima, nell'ambito *healthcare*, il *sensor* era rappresentato dal sensore di monitoraggio del glucosio nel sangue. Un *sensor* può avere un'identità all'interno del *cluster* di sensori laddove vi è necessità di differenziare la provenienza dei dati raccolti [4].
Gli *aggregator* invece operano ad uno step successivo rispetto ai *sensor*, infatti, questi si occupano di aggregare i dati, talvolta *big data*, aiutandone il processamento successivo. In linea di massima, un *aggregator* può essere visto come una implementazione software di una funzione matematica, per questo motivo, richiedono potenza computazionale. Per poter aggregare i dati, utilizzano i *clusters* e i *weights*, che sono rispettivamente un'astrazione di un gruppo di sensori in grado di attivarsi e disattivarsi uniformemente, mentre i *weights* rappresentano quanto un *sensor* effettivamente impatterà sul *aggregator* preso in considerazione. I *weights* possono avere valori dinamici o statici, nel secondo caso sono definiti all'inizializzazione del sistema. È consigliato un valore di *weight* configurabile durante tutto il ciclo di vita

del sistema laddove l'ambiente sia dinamico e di conseguenza i sensori assumono importanza variabile nel tempo. La modifica dei *cluster* e dei *weights* in un modello dinamico può essere realizzata con tecniche di intelligenza artificiale. Un banale esempio di *aggregator* potrebbe essere quello che, partendo da un *cluster* di sensori, calcola la media pesata di ogni *sensor* utilizzando i vari *weights* come pesi per effettuare la computazione.

I *communication channel* rappresentano invece l'astrazione del canale di comunicazione attraverso il quale i dati vengono trasmessi, può essere fisica o non (e.g wireless). Ricordando che *IoT* opera su flussi di dati si possono considerare le *external utility* e i *decision trigger* come la parte finale di quella che può essere considerata un'astrazione di *pipeline*. In particolare, le *external utilities* rappresentano un servizio (hardware o software) in grado di far eseguire dei processi, per esempio, si possono supporre locate nel cloud con il compito di eseguire alcuni *aggregator*. Infine, i *decision trigger* si occupano di fornire quello che sarà il risultato finale. Può essere estremizzata la sua rappresentazione in una espressione booleana che permette di decidere se effettuare o meno una determinata azione (e.g. accensione del riscaldamento in seguito alla temperatura rilevata, o rimanendo nell'esempio prima illustrato, decidere se iniettare o meno l'insulina nel corpo del paziente).

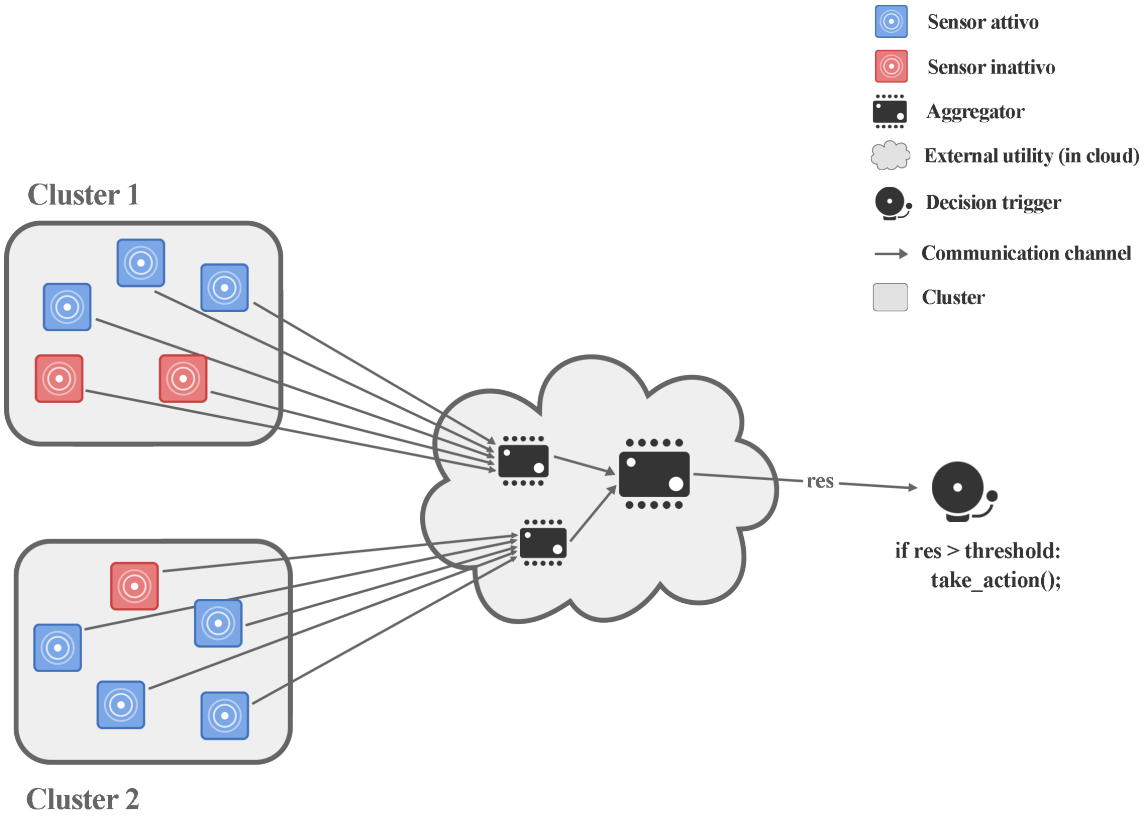

Figura 1.1: *Schema IoT*

1.1.2 Industrial Internet of Things

Uno dei settori principalmente influenzati dall'*IoT* è sicuramente quello dell'industria manifatturiera, in seguito all'avvento di questa tecnologia l'industria ha subito un inevitabile trasformazione in quella che è ormai conosciuta come Industria 4.0. Introducendo sfide come l'analisi di *big data*, il *cloud*, la robotica, ecc., si può affermare che *IoT* sia per eccellenza la tecnologia in grado di rivoluzionare questo campo, unendo aspetti informatici con aspetti di produzione. Quando si parla di industria manufatturiera si può considerare come principale beneficio il miglioramento di alcuni processi produttivi attraverso l'analisi di dati. Si stima infatti che in futuro, attraverso questo approccio, sempre più aziende saranno in grado di generare miglioramenti sia dal punto di vista produttivo che di modello di business [15].

È chiaro come i dati generati dai sensori assumano un ruolo fondamentale, questi dati possono essere impiegati per diversi scopi, tra cui il monitoraggio di sistemi. Altri benefici forniti dall'*I-IoT* comprendono l'asset management riuscendo a monitorare la qualità, le performance, potenziali danni, colli di bottiglia, ecc. e forse

quello che più tange in maniera diretta l'industria, l'ottimizzazione della produzione stessa, riducendo costi e sprechi. Se si volessero riassumere quali potrebbero essere alcuni tra i principali benefici che si otterrebbero dall'analisi di dati generati da sempre più sensori nell'industria, sicuramente si troverebbero:

- Diagnostica e previsione di eventuali guasti
- Manutenzione
- Efficienza nella produzione
- Minori consumi e produzione di scarti

Nel caso di analisi di tipo *data-driven*, in opposizione al modello che utilizza formule matematiche per l'analisi, diventa quasi indispensabile introdurre modelli di machine learning per migliorare soprattutto l'accuratezza delle analisi stesse. Inoltre, con il prendere piede del *deep learning*, branca dell'intelligenza artificiale, sempre più modelli *data-driven* sono presi in considerazione.

1.2 Cloud computing

Il *cloud computing* fornisce la possibilità di usufruire di potenza computazionale e/o di memoria attraverso risorse fisicamente distanti. Fornisce una virtuale illimitatezza nell'utilizzo di risorse che esse siano computazionali o di memoria, sopperendo ad esempio, alle limitazioni che possono avere dispositivi facenti parte di un'infrastruttura *IoT*. Il *cloud* in opposizione a un *datacenter* fisico e centralizzato presenta diversi vantaggi, non solo dal punto di vista economico, ma anche dal punto di vista della gestione e della scalabilità. In un *datacenter* auto-gestito per eseguire operazioni di scalabilità è necessario un intervento economico non indifferente, sia per l'acquisto di hardware che per la manutenzione dello stesso e per l'assunzione di personale qualificato. Inoltre, necessita di spazio fisico dove poter mantenere i server, con conseguente gestione dello stabile. Un'altra caratteristica importante del cloud computing è che permette di non doversi preoccupare della gestione della sicurezza e del recupero dei dati nel caso di guasto.

1.2.1 Edge Cloud Computing

In alcuni contesti come, ad esempio, nell'*I-IoT*, Sezione 1.1.2 , oppure laddove è richiesto un tempo di risposta a bassissima latenza o real-time, il classico approccio al *cloud computing* potrebbe non essere efficace. Basti pensare in uno scenario di Industria 4.0 dove alcuni sensori possano rilevare un guasto e come questo abbia bisogno di un'azione immediata per essere corretto ed evitare la compromissione

dell'intero sistema. L'*edge cloud computing* (o *edge computing*) risolve questi problemi, risultando un'ottima soluzione in sistemi dove la bassa latenza sia un rquisito. Si può affermare che l'*edge computing* operi vicino alle sorgenti di dati utilizzando piccoli *edge datacenter*, mentre le operazioni più onerose e che non richiedono una bassa latenza di risposta possono essere effettuate normalmente nel *cloud*. Un altro aspetto da tenere in considerazione è l'utilizzo di banda, con una computazione vicina alla sorgente di dati si eviterebbe di inviare i dati grezzi che necessitano di essere processati. L'*edge computing* diventa sempre più importante a causa della crescita nella produzione dei dati dai numerosissimi endpoint di varia natura, uno spostamento di dati risulterà sempre più oneroso, per cui diventerà sempre più importante migrare la computazione verso la sorgente dei dati. In un certo senso si può affermare che l'*edge computing* sposti il modello di esecuzione dall'essere centralizzato ad un modello di esecuzione distribuito e decentralizzato. Tuttavia, utilizzando un modello decentralizzato è richiesta una maggiore attività di monitoraggio e controllo sui diversi nodi, aumentando inevitabilmente la complessità del sistema e la sua manutenzione.

Volendo riassumere quali possono essere i benefici dell'*edge computing* oltre a quelli già citati come la bassa latenza di risposta, il basso utilizzo di banda e la minore possibilità di congestione della rete, si possono anche apprezzare importanti gradi di autonomia in sistemi dove ci sono forti limitazioni sull'utilizzo di banda. Operando infatti in locale è permesso di trasmettere i dati già processati (significativamente ridimensionati), oppure in attesa che la connessione sia ristabilita in luoghi con scarsa connettività. Risolve anche problemi legati alla sicurezza, alla privacy, infatti, evitando di spostare grandi quantità di dati si riduce la possibilità di un *data leak*. Dall'altra parte però la potenza computazionale di un *edge device* spesso non è minimamente paragonabile a quella di un server nel cloud e spesso, carente di sistemi di sicurezza interni come firewall o altri sistemi di protezione. Anche l'eterogeneità dei dispositivi necessità di comunicazioni diverse con protocolli differenti rendendo la gestione della sicurezza un compito complesso [11].

In conclusione, gli scenari dove è particolarmente apprezzato l'uso dell'edge computing sono [13]:

- Necessità di estrarre valore dall'analisi di dati in maniera veloce ed efficiente
- Riduzione dei costi dovuti al trasferimento dei dati sulla rete
- Riduzione della vulnerabilità in termini di sicurezza
- Controllo della privacy

1.3 Serverless

Serverless, anche conosciuto come *FaaS* (Function as a Service) [14], non è come può sembrare dal nome, un modello dove non è presente il server, si tratta invece, di un modello di esecuzione in cui l'allocazione delle risorse e tutto ciò che è concerne la gestione del server stesso, viene demandata direttamente al *cloud provider*, a differenza ad esempio, del modello *IaaS*, dove comunque vi è la necessità di gestione dell'infrastruttura da parte dello sviluppatore. Il termine gestione racchiude diverse azioni che devono invece essere effettuate in un modello *client-server* comune, alcune di queste possono essere il *deployment*, il setup dell'ambiente, la manutenzione, la gestione della scalabilità, la gestione della sicurezza, *monitoring*, ecc. Grazie al modello *Function as a Service* la possibilità di creare applicazioni con caratteristiche importanti di scalabilità è diventata più rapida e semplificata. Da non sottovalutare anche l'aspetto economico e di utilizzo di risorse, infatti, essendo un modello *on-demand*, l'allocazione di risorse è effettuata solo alla richiesta, conseguentemente il costo sarà direttamente proporzionale all'utilizzo del servizio stesso [9].

Questo comportamento è permesso grazie al fatto che Serverless sia un modello *event-driven*, un pattern dove il servitore risponde sulla base di eventi ricevuti, permettendo tra le varie cose, un accoppiamento debole tra i nodi dell'infrastruttura. Le richieste API sono un esempio molto utilizzato di sistemi *event-driven*.

Con il modello introdotto diventa inoltre possibile definire più funzioni per richieste diverse, introducendo anche la possibilità di avere diversi server che rispondono ad esse e di conseguenza più database a cui essi si interfacciano (potenzialmente attraverso un partizionamento, verticale o orizzontale del database stesso). Ricordando che i server sono resi attivi e inattivi sulla base di eventi generati direttamente dalle richieste dell'utente, è immediato capire come la gestione della scalabilità risulti semplificata, considerando infatti la separazione delle funzioni implementate si può decidere dove optare per un'operazione di scalabilità, permettendo di agire localmente e non scalando l'intero sistema come invece avverrebbe per un'applicazione monolitica. In Figura 1.3 e in Figura 1.2 si possono vedere le differenze tra un classico modello *client-server* e un modello Serverless. Si può notare la separazione tra funzioni diverse, con conseguente allocazione di server diversi e con il rispettivo interfacciamento sui database di interesse.

In particolare, in Figura 1.3 si può supporre che la prima funzione abbia un numero più elevato di richieste, in quanto ipoteticamente, il sistema abbia dovuto eseguire delle operazioni di scalabilità aumentando il numero di servitori allocati. Banalmente nel caso del modello *client-server* il blocco centrale, che riceve richieste

dal client e possibilmente si interfaccia al database per fornire una risposta, non gode di nessuna delle caratteristiche sopracitate.

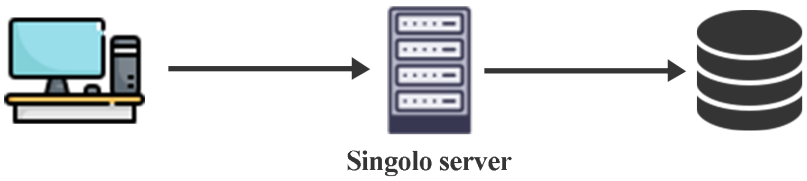

Figura 1.2: *Schema classico client-server*

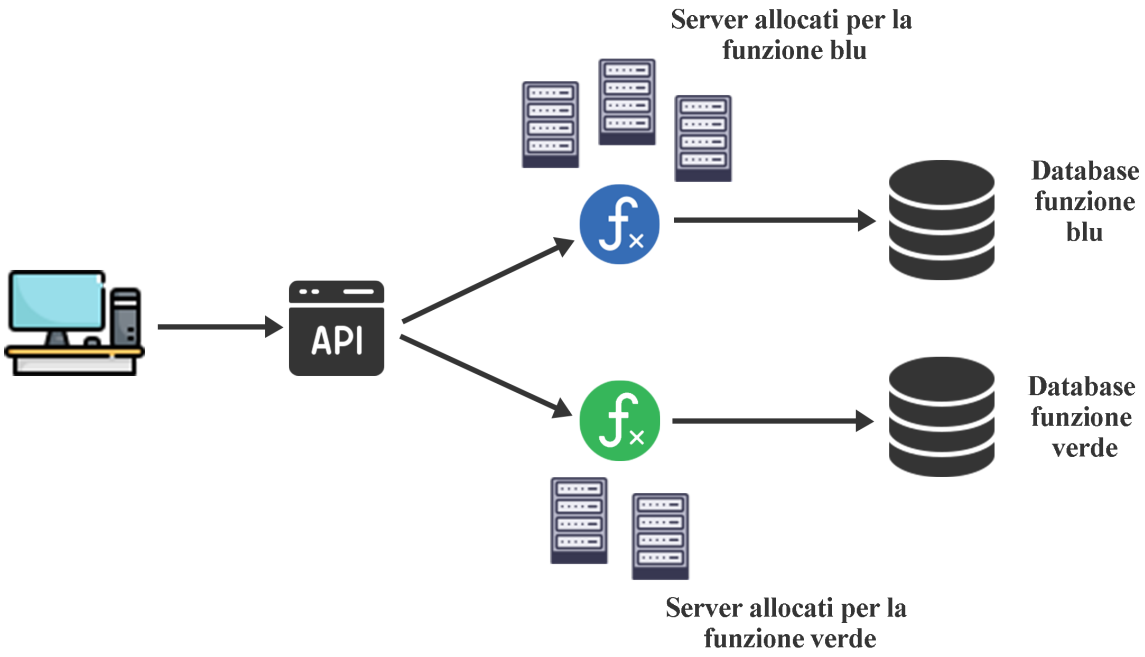

Figura 1.3: *Schema di un modello Serverless*

1.3.1 Vantaggi e svantaggi

Tra le cose che più saltano all'occhio sicuramente abbiamo la facilità d'uso, direttamente collegata con la velocità di sviluppo. Infatti, grazie ai servizi forniti dai cloud provider, lo sviluppatore deve solo preoccuparsi di fornire la logica di business dell'applicazione, senza preoccuparsi di come questa viene gestita, mantenuta e fornita all'utente finale. È dunque inevitabile un aumento della facilità d'uso e una drastica riduzione del tempo di sviluppo. Tra questi servizi si ricorda ancora

che vi è la scalabilità, direttamente collegata con un'altra caratteristica importante e non ancora citata che è la disponibilità, spesso infatti, sistemi di questa natura richiedono tempi di disponibilità estremamente alti. La possibilità di allocazione dinamica delle risorse sulla base di richieste permette di rispondere ad entrambe le necessità mantenendo quindi un servizio efficiente e disponibile con tempi di *down* estremamente bassi. Non bisogna tuttavia trascurare l'aspetto economico, che elencato tra i vantaggi, risulta essere relativamente basso (prendendo in considerazione i principali *provider* sul mercato).

Tuttavia, anche Serverless presenta alcuni svantaggi, o forse meglio chiamarli limiti. Primo fra tutti lo sviluppatore non ha il pieno controllo del sistema, nonostante si possa supporre che il sistema sia completamente gestito dal *provider*, alcune volte risulta necessario avere una visione dello stesso per effettuare qualche modifica sulla base di necessità che si possono presentare anche in fase di produzione. Anche se, solitamente i *cloud provider* forniscono il controllo su alcuni parametri come può essere l'allocazione delle risorse computazionali o di memoria. Serverless risulta infatti sconsigliato nel caso in cui, per qualche motivo, si ha la necessità di avere un controllo pressoché totale sull'infrastruttura. Un altro limite che può non far propendere per una soluzione di questo tipo può essere di natura commerciale, in particolare, si pensa che la scelta di un determinato *provider* sia vincolante, in realtà ci sono diverse tecniche per evitare che questo accada e poter quindi facilmente migrare su altri *provider*, ad esempio, astraendo la logica relativa allo specifico *cloud provider*.

Un ultimo svantaggio/limite che verrà mostrato, preoccupa maggiormente gli sviluppatori di sistemi ad alta efficienza anche con richieste sparse nel tempo. In generale, dopo un determinato tempo (cinque minuti solitamente) se un processo allocato non dovesse ricevere richieste verrebbe terminato. Questo può fare incorrere nel problema del "cold start", ovvero quando il sistema deve riallocare le risorse per servire le richieste e di conseguenza con un tempo di risposta maggiore. Nonostante ciò, se si conosce a priori il comportamento delle richieste nel tempo si può risolvere utilizzando uno scheduler.

Capitolo 2

Tempos

Contents

In questo capitolo si introdurrà il *middleware* TEMPOS e se ne vedranno le *performance* prima dell'estensione oggetto di questa tesi. TEMPOS (*Time-Effective Middleware for Priority Oriented Serverless*) si presenta come un *middleware* orientato all'*Internet of Things* con lo scopo di gestire diversi aspetti della qualità del servizio (*QoS*) di una moltitudine di risorse virtualizzate, il tutto in ambienti con risorse *edge cloud* per poter implementare il modello di esecuzione *serverless* nel *cloud continuum*. Il controllo sulla qualità del servizio è richiesto in diversi settori, anche come conseguenza dell'eterogeneità degli effetti causati, per esempio, da malfunzionamenti. L'industria manufatturiera rappresenta un perfetto esempio di ambienti relativi a questa tipologia. Lo scambio di informazioni, in sistemi caratterizzati da diversi componenti, spesso richiede che l'informazione venga trattata con diversi gradi di priorità e conseguentemente con diversi livelli di qualità del servizio. Quello che TEMPOS propone è l'esecuzione di una piattaforma *serverless*, *Function as a Service*, che esegue dinamicamente funzioni all'arrivo di determinati eventi, eventi che possono spaziare da una semplice richiesta HTTP a un evento generato come conseguenza del superamento di una soglia in un sistema di sensori facenti parte

di un sistema *IoT*. Tuttavia, la parte che contraddistingue TEMPOS è la gestione delle priorità riguardante la consegna dei messaggi e la conseguente esecuzione delle funzioni associate. A tal proposito, senza perdere di generalità, TEMPOS sfrutta lo *scheduling real-time* di Linux, priorità differenziate su un *Message-oriented middleware (MoM)* e il *Time-sensitive networking* (TSN) [3].

2.1 Architettura

La tipica struttura di un *FaaS* consiste in tre componenti: *Trigger*, *Controller* ed *Executor*. Il *Trigger* è il componente che si interfaccia all'esterno, ricevendo segnali e trasformandoli in modo da permettere l'esecuzione di determinate funzioni basate su di essi. Il *Controller* è responsabile, oltre che alla configurazione e gestione del ciclo di vita degli altri componenti appartenenti al *middleware*, della gestione degli eventi che il *Trigger* fornisce, in seguito alla trasformazione dell'evento stesso. Il *Controller* in seguito alla ricezione di un segnale può comunicare con gli *Executors*, in maniera sincrona (chiamata a funzione) o asincrone (relazione *publish-subscribe*), per richiedere che l'evento venga processato. Gli *Executors*, invece, sono installati in nodi fisici e hanno il compito di contenere ed eseguire funzioni in seguito alla ricezione di eventi da parte del *Controller*. In particolare, il processo responsabile di questo comportamento è chiamato *Invoker* o *Watchdog*. Per sopperire alla moltitudine di scenari applicativi TEMPOS è stato suddiviso in sezioni differenti: *Bridging*, *Delivery* e *Processing*, più un *Controller* con compito di gestione delle diverse sezioni. La comunicazione tra le diverse sezioni è ottenuta tramite delle interfacce standardizzate che permettono la genericità dei protocolli e delle tecnologie adottate.

2.1.1 Controller

Il *Controller* in TEMPOS oltre che avere il compito di gestire le varie sezioni ha la funzione di gestire i *workflow* definiti dai clienti, fornendo una interfaccia semplificata degli *endpoint* per, ad esempio, mappare le diverse *QoS* richieste dal cliente. In particolare, riceve le informazioni per effettuare le configurazioni delle parti che compongono l'infrastruttura e l'insieme di *workflow* con annessi i livelli di priorità. Grazie alla sincronicità della comunicazione tra il *Controller* e i vari componenti è possibile anche una configurazione *run-time*.

2.1.2 Delivery

Questa sezione è responsabile della distribuzione degli eventi (eventualmente caratterizzati da *QoS* differenti) ai vari componenti di TEMPOS. Per ottenere un

simile comportamento viene impiegato un *MoM* (*Message-oriented Middleware*) e attraverso la creazione di un canale i componenti del *middleware* possono collegarsi in modo bidirezionale al *MoM* stesso. Grazie a degli *Adapter*, inoltre, il *MoM* può collegarsi a canali di diverso tipo in maniera omogenea, comprendo così anche situazioni di contesti eterogenei. La capacità di gestire messaggi con diverse priorità deriva dalla concezione di code di messaggi con molteplici priorità, così facendo la gestione dei messaggi avviene per ordine di priorità. Come astrazione della qualità viene introdotto il concetto di *Topic*, caratterizzato da un insieme di canali in ingresso e uscita e una coda di priorità. È il costrutto che permette di astrarre i diversi livelli di *QoS* e a cui ci si può iscrivere per ricevere i messaggi che passano per quel determinato *Topic*. Inoltre, con il meccanismo dei *Subscription Groups* si ottiene un comportamento di *load balancing*, necessario per un *MoM* in contesto di *FaaS*, in quanto i messaggi ricevuti dal *Topic* stesso vengono mandati una sola volta e ad un solo membro del *Subscription Group*.

2.1.3 Bridging

La sezione relativa al *Bridging* permette di unificare le richieste ricevute, trasformando gli eventi in una rappresentazione interna unificata che permette la gestione da parte di tutte le sezioni del *middleware*. Il componente principale è chiamato *Trigger* e ha il compito di trasformare in eventi i segnali ricevuti, per poi inoltrarli al *MoM*. Fornisce all'esterno diversi *endpoint* per i diversi livelli di QoS, con il compito di tradurli conseguentemente nella rappresentazione interna. Esistono tre diversi scenari relativi al *deployment* del *Trigger*. Il primo vede il *Trigger* co-locato con la sorgente esterna degli eventi, la seconda vede invece il *Trigger* è posizionato tra la sorgente e il *MoM*, mentre la terza opzione vede il *Trigger* co-locato con il *MoM*. Sostanzialmente nel primo caso si può vedere come il *Trigger* sia quasi usato come risorsa dedicata a una determinata sorgente di eventi, rendendo difficile l'utilizzo dello stesso *Trigger* per sorgenti diverse, svantaggio che non si ha nel secondo caso, tuttavia nella seconda configurazione vi è la possibilità di avere conflitti nel caso in cui si richieda la stessa *QoS*, nell'ultimo caso, più particolare in termini di scenario, la sorgente deve fornire segnali già in forma di evento e quindi in questo caso TEMPOS farà parte di un'infrastruttura pre-esistente e che potenzialmente già comunica attraverso lo scambio di eventi.

2.1.4 Processing

Questa sezione si può considerare come la parte terminale di TEMPOS ed ha il compito di processare gli eventi che la sezione *Delivery* fornisce. Permette al cliente di specificare la logica di business e la *QoS* richiesta, astraendo l'implementazione

delle stesse. Il costrutto responsabile dell'esecuzione è chiamato *Invoker*, incaricato di istanziare la funzione specificata dal cliente e propagare l'evento ad essa.

2.2 Implementazione

In questa sezione si farà riferimento a TEMPOS dal punto di vista implementativo. Per quanto riguarda il supporto alla *QoS*, TEMPOS fornisce due diverse opzioni, la prima, detta anche *Best-Effort Quality (BQ)*, è caratterizzata da vincoli non molto stringenti per quanto riguarda i tempi di latenza e di jitter per l'invocazione di funzioni o di scambio di messaggi, la seconda, *Strict Quality (SQ)*, con vincoli più stringenti e con tempi di risposta *soft real-time*. Il primo componente che verrà introdotto sarà il *Controller*.

Il *Controller* è stato implementato come un processo demone *Linux* con il compito di inizializzare l'intero sistema (si ricorda che ne ha una visione complessiva grazie a una rappresentazione interna)Per farlo utilizza un file di configurazione *TOML* che specifica i protocolli di comunicazione con i vari componenti del middleware e le informazioni relative alla *QoS* richiesta per la comunicazione e per l'esecuzione delle funzioni. Alla ricezione di richieste tramite *API REST* il *Controller* è in grado di riconfigurarsi interagendo in maniera sincrona con gli altri componenti.

Il *Message-oriented Middleware*, facente parte della sezione *Delivery*, come anticipato prima sfrutta il costrutto di canale di comunicazione, in particolare mantiene due code di messaggi, gestite da due *thread* separati e implementate come *socket* di rete, ognuna relativa alle diverse *QoS*. La schedulazione delle stesse avviene tramite il *real-time scheduler* di *Linux*. Attraverso il *Controller* si possano specificare configurazioni differenti per lo *scheduling*, di base il valore di priorità 0, fornito dallo *scheduler*, rappresenta la *QoS Best-Effort Quality*, mentre il valore di priorità 99 rappresenta la *Qos Strict Quality*. Per la gestione dell'eterogeneità dei contesti e per fornire un buon grado di trasparenza viene introdotto il concetto di *Adaptor*. Permette di gestire in maniera uniforme un elevato numero di canali eterogenei ed è realizzato attraverso l'utilizzo di un insieme di interfacce che definiscono come stabilire una connessione, configurare la *QoS* richiesta, mandare messaggi attraverso il canale e come chiudere correttamente la connessione. Questo sistema viene implementato attraverso il meccanismo di caricamento dinamico delle librerie, infatti, il *Message-oriented Middleware*, in base ai messaggi di configurazione ricevuti dal *Controller*, ha il compito di caricarle a *run-time*. Il plug-in è bastato sugli standard *TSN (Time Sensitive Networking)*, in particolare sullo standard *IEEE 802.1Qbv*, che supporta traffico di tipo *best-effort* e *real-time*. Per minimizzare l'interferenza del traffico non prioritario con il traffico con *QoS* di tipo *Strict-Quality*, lo standard

introduce la nozione di *time-triggered communication window* che supporta diversi flussi per richieste di tempi di risposta diversi. Grazie a una *guard band* prima della *window* appena citata si può ottenere il buffering dei pacchetti che in base alla classe di appartenenza non dovranno essere trasmessi, potendo così prioritizzare gli altri.

Il *Trigger* è il componente appartenente alla sezione *Bridging* e si occupa di trasmettere al *Messsage-oriented Middleware* gli eventi generati dalle sorgenti. È implementato come un processo *Linux* continuamente in ascolto su una *socket*. All'esecuzione riceve dal *Controller* la configurazione per la *QoS* richiesta, per poi inoltrare l'evento al *MoM*. I *Trigger* hanno un'implementazione limitata a un solo protocollo che permette la prioritizzazione (per i *Trigger* co-locati, ad esempio si sfrutta il meccanismo di comunicazione basato su *TSN*).

L'*Invoker*, parte della sezione *Processing*, è implementato come un componente *multi-threaded* con il compito di gestire due *thread*, il primo per la gestione delle richieste di configurazione ricevute dal *Controller* (grazie alla quale imposta la *QoS*), il secondo per la gestione delle richieste effettive di chiamate delle funzioni. In questo caso lo sviluppatore oltre che dover specificare la *QoS* richiesta dovrà specificare anche quale tra queste tre modalità di esecuzione delle funzioni sarà utilizzate:

- ***DLF (Dynamically Loaded Function)***: questa modalità permette di risparmiare memoria primaria e secondaria grazie alla tecnica di caricamento dinamico delle librerie che permette a più processi di condividere un insieme di funzioni. Per richiedere la chiamata di una funzione lo sviluppatore deve aver esposto la signature della funzione nella libreria, al momento della richiesta di chiamata a funzione l'*Invoker* apre e carica l'oggetto relativo alla libreria. Successivamente la funzione viene eseguita e scaricata.

- ***WASMF (WASM Function)***: in questo caso viene integrato un motore *WASM* in grado di caricare dinamicamente la libreria condivisa contenente la funzione richiesta, la libreria deve però essere compilata usando un *WASM code generator* in grado di tradurre una rappresentazione intermedia indipendente dal target. Grazie a questo meccanismo e alla traduzione dinamica, l'implementazione della funzione può essere effettuata in diversi linguaggi di programmazione mantenendo comunque l'efficienza.

- ***FSpawn (Function Spawn)***: l'ultimo caso, è forse quello più classico e vede l'utilizzo di *posix_spawn()* di *Linux*, nel caso in cui non sia supportato quello che viene utilizzato saranno in sequenza una *fork()*, per creare un processo figlio, e una *exec()* per eseguire il programma.

2.3 Performance

In questa sezione si mostreranno ed analizzeranno alcuni dei test effettuati precedentemente allo sviluppo di questa tesi.

2.3.1 Descrizione testbed

I testbed saranno effettuati per validare l'efficacia di TEMPOS nel gestire diverse richieste di *QoS* e per numeri di richieste in grado di mettere sotto stress il sistema. Saranno pensati per mettere sotto stress alternativamente il processo responsabile della consegna degli eventi, del loro processamento e del middleware nel complesso. Nel primo caso si emuleranno diversi *workflow* sotto diverse condizioni di carico, garantendo il rispetto per le richieste di latenza e di *jitter*. Nel secondo caso si vuole mostrare la capacità di TEMPOS di fornire una differenziazione della *QoS* nascondendo l'eterogeneità attraverso la chiamata a una funzione computazionalmente pesante pur mantenendo la disponibilità del servizio.

Algorithm 1 Pseudo codice della funzione utilizzata nei test: deserializzazione dell'evento, conteggio delle occorrenze nel testo e ripetizione delle funzioni matematiche potenza e radice quadrata.

```
function MAIN(e : Event)
    message, pattern ← deserialize(e)
    occur ← count_occurence(message, pattern)
    res ← 0
    for i ← 0, occur do
        if i mod 2 = 0 then
            res ← res + pow(i)
        else
            res ← res + sqrt(i)
        end if
    end for
    output(res)
end function
```

Nel terzo e ultimo caso si vuole invece testare la capacità di TEMPOS di generare meccanismi per la gestione della *QoS* nelle diverse sezioni. Si utilizzano in particolare le due *QoS*, *Strict-Quality* e *Best-Effort Quality*, generando due *workflow* distinti che invocano la funzione nell'Algoritmo 1. Per i test riguardanti il deployment nel caso di *edge cloud computing*, dove le risorse sono più limitate, si sono utilizzati nodi con capacità computazionale ridotta. Gli *host edge* sono rappresentati da tre nodi *TSN*. Per la selezione della *QoS* si utilizzano due modalità di accodamento del *kernel Linux*, la prima *taprio (Time-Aware Priority Shaper)* che implementa una versione semplificata dello standard *IEEE 802.1Qbv*, la seconda *etf (Earliest TxTi-*

me First) che permette di specificare un tempo di trasmissione per i pacchetti. È stata inoltre introdotta un'estensione del *Precision Time Protocol (PTP)* chiamata *generic Precsion Time Protocol (gPTP)* in quanto le applicazioni basate sullo standard *IEEE 802.1Qbv* devono fare affidamento a un unico riferimento temporale. Ci sono due entità in gioco i *Clock Master (CM)* e i *Clock Slave (CS)*, inoltre esiste un nodo *grandmaster PTP* determinato da un algoritmo, *Best Master Clock Algorithm (BMCA)* tra i vari *Clock Master*, in grado di mandare informazioni sul clock a tutti i *Clock Slave* in modo che tutti i componenti di TEMPOS siano sincronizzati. Sono state quindi create due finestre temporali *TSN* di 1 ms ciascuna, la prima tra *Trigger* e *Message-oriented Middleware*, mentre la seconda tra *Invoker* e *Message-oriented Middleware*. Ogni finestra è a sua volta divisa in 2 slot temporali di 0.5 ms ciascuna. Per il tracciamento delle performance viene eseguito il log dell'id di ogni evento con il relativo *timestamp*.

2.3.2 Risultati e performance

Per quanto riguarda la delivery degli eventi, nel primo test si sono generati al *Trigger* 1000 eventi al secondo per 5 minuti confrontando la differenza di tempo tra la generazione dell'evento e il *timestamp* relativo alla consegna dell'evento stesso da parte del *Trigger*, si può notare in Figura 2.1 come la latenza introdotta per la *Best-Effort Quality* sia molto maggiore rispetto alla *Strict Quality*. Una volta trasmesso dall'*Trigger*, un evento raggiunge l'*Invoker* in non più di 0.7 ms, in particolare nel peggiore dei casi si è visto che i tempi di consegna tra i vari componenti sono:

- 223 µs: consegna evento al *Message-oriented Middleware*
- 57 µs: processamento dell'evento
- 299 µs: consegna dell'evento all'*Invoker*

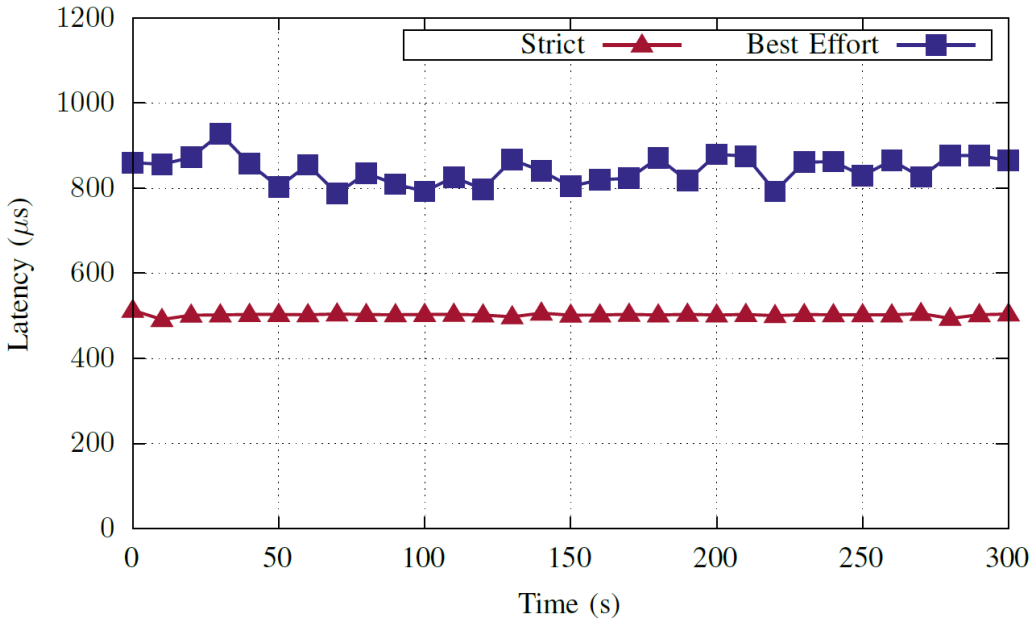

Figura 2.1: *Test sulla latenza della sezione delivery con QoS diverse quando eseguite separatamente*

Il secondo test è stato eseguito in un contesto con più *workflow* attivi e quindi con la necessità di condividere risorse, tra *workflow* anche con *QoS* differenti. Dapprima mandando in esecuzione 1 *workflow* per ogni *QoS* supportata e poi aumentando a 3 il numero di *workflow* della *Best-Effort Quality*. Anche qui si sono generati 1000 eventi al secondo per 5 minuti. Si può notare in Figura 2.2 come la *Strict Quality* non sia affetta dall'esecuzione dei *workflow* in modalità *Best-Effort Quality*. Il *jitter* che si ottiene è a causa dell'utilizzo di un dispositivo per l'elaborazione di pacchetti in grado di migliorare le prestazioni (*NAPI*), in grado di ridurre i tempi di latenza, aumentando però il *jitter* e l'utilizzo della CPU.

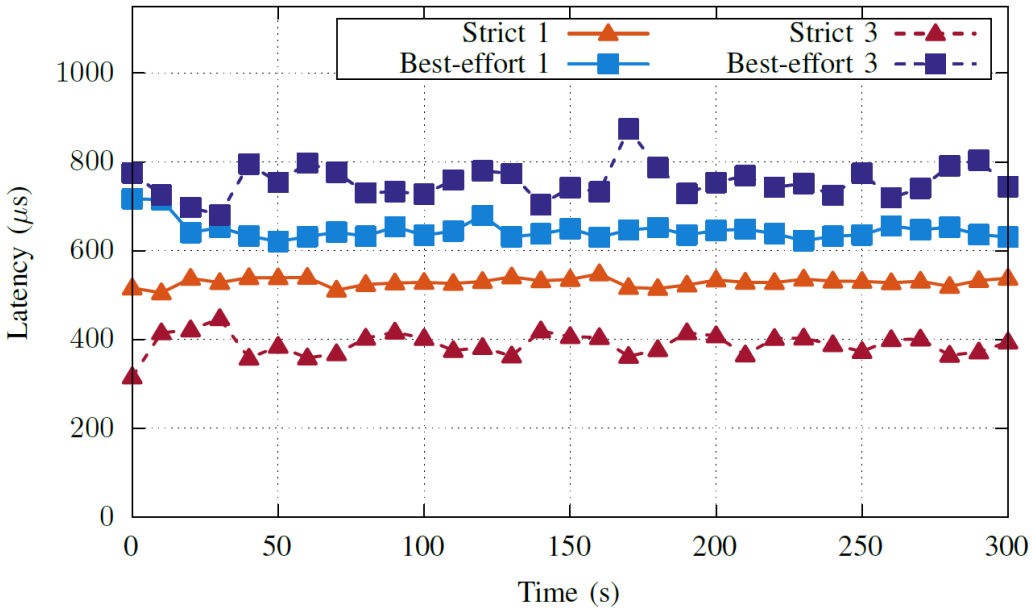

Figura 2.2: *Test sulla latenza della sezione delivery con QoS diverse quando eseguite concorrentemente, prima con 1 SQ e 1 BQ poi con 1 SQ e 3 BQ*

Per il processamento interno degli eventi, nel primo test si è invocata la funzione nell'Algoritmo 1 per due minuti nelle diverse modalità di esecuzione della funzione, prima con un linguaggio compilato poi con linguaggi interpretati (*Python* e *Javascript*). Il test è stato effettuato su 3 diversi nodi con risorse differenti evincendone che: i tempi di esecuzione e di *startup* sono influenzati dall'*hardware* utilizzato e che latenze inferiori a 1 ms sono ottenibili in *hardware* di fascia medio-alta. La modalità di esecuzione *DLF* nonostante i limiti sul linguaggio utilizzato risulta la più efficiente (circa 0.1 ms). La tipologia *FSpawn* invece risulta quella che offre la maggiore flessibilità in termini di linguaggio, rendendola adatta per il contesto *FaaS*, ma le peggiori prestazioni. Per quanto concerne la modalità *WASMF* si comporta leggermente meglio, in termini di latenza della modalità *FSpawn* compilata, ma comunque inferiore alla tipologia *DLF*. In questo caso è interessante notare che si può ottenere un sensibile miglioramento nei tempi di esecuzione e *startup* di alcuni linguaggi non compilati. Nel secondo test si è paragonata l'esecuzione delle tre tipologie ma in un contesto concorrente di 2 funzioni in *Strict Quality* e 4 funzioni in *Best-Effort Quality*. I risultati sono visibili in Figura 2.3 ed è visibile che i tempi sono circa dimezzati per la *SQ*. Inoltre, per quest'ultima modalità si può apprezzare un tempo di latenza costante, rendendola ideale nei casi in cui sia importante prevedere i tempi di risposta o laddove è importante avere tempi di risposta costanti.

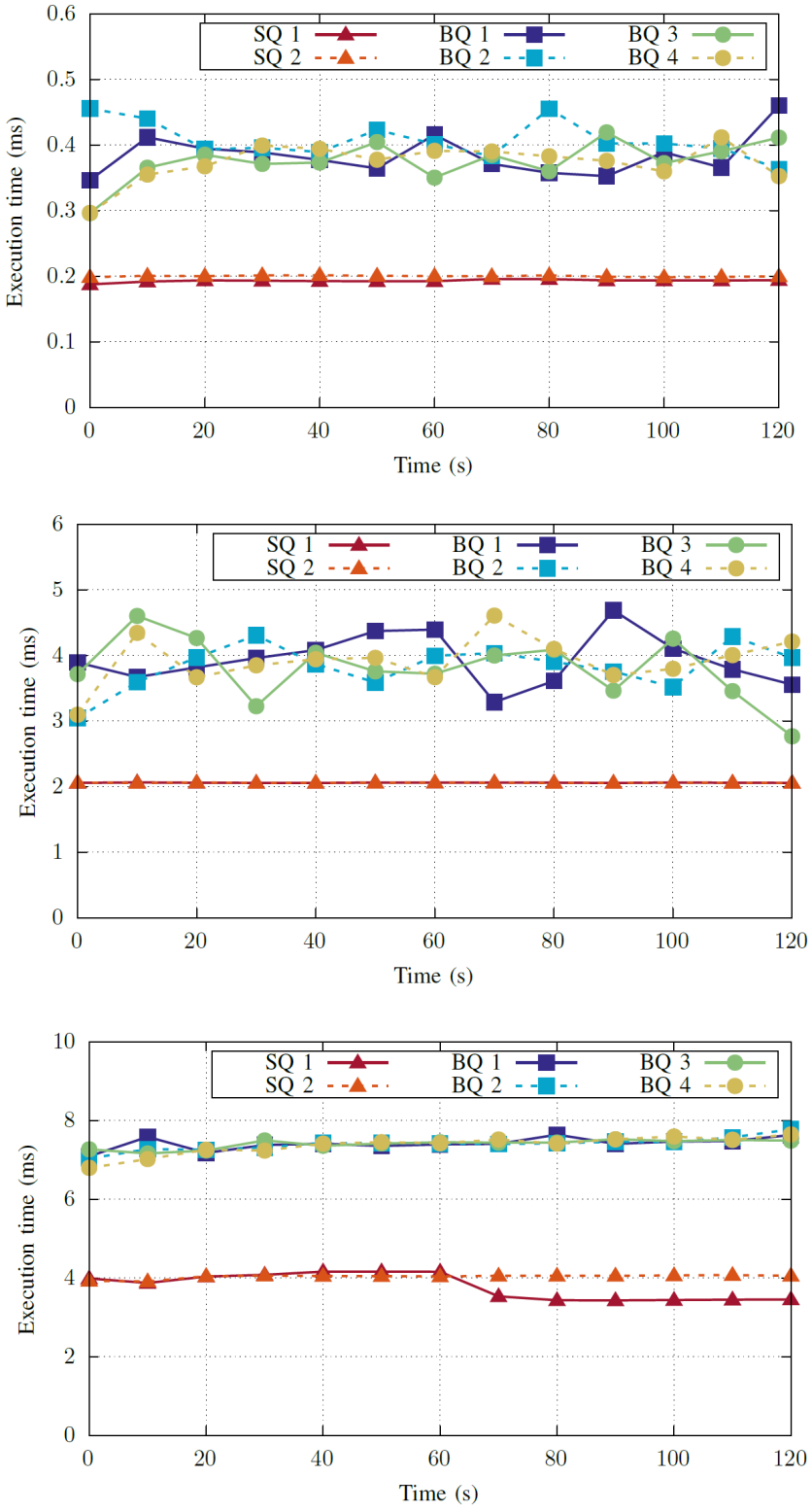

Figura 2.3: *Test delle funzioni eseguite concorrentemente configurate con QoS differenti, dall'alto al basso: Dynamic Loaded Function, Function Spawn e WASM Function*

Infine, per testare la capacità di TEMPOS di coordinare e concatenare tra le varie sezioni le diverse QoS si sono creati due *workflow* innescati dallo stesso evento. Il test è stato effettuato aumentando in maniera lineare gli eventi generati fino ad arrivare a 1000 eventi generati al secondo per ogni *workflow*. Inizialmente i due *workflow* operano distintamente e successivamente in maniera concorrente. Come si può vedere in Figura 2.4 la latenza relativa alla *Strict Quality* si comporta meglio del caso *Best-Effort Quality*, questo risulta più evidente quando si raggiungono circa i 500 eventi generati al secondo dove si può apprezzare un importante degrado delle prestazioni nel caso *BQ*, mentre *SQ* rimane pressoché invariata in quanto prioritaria.

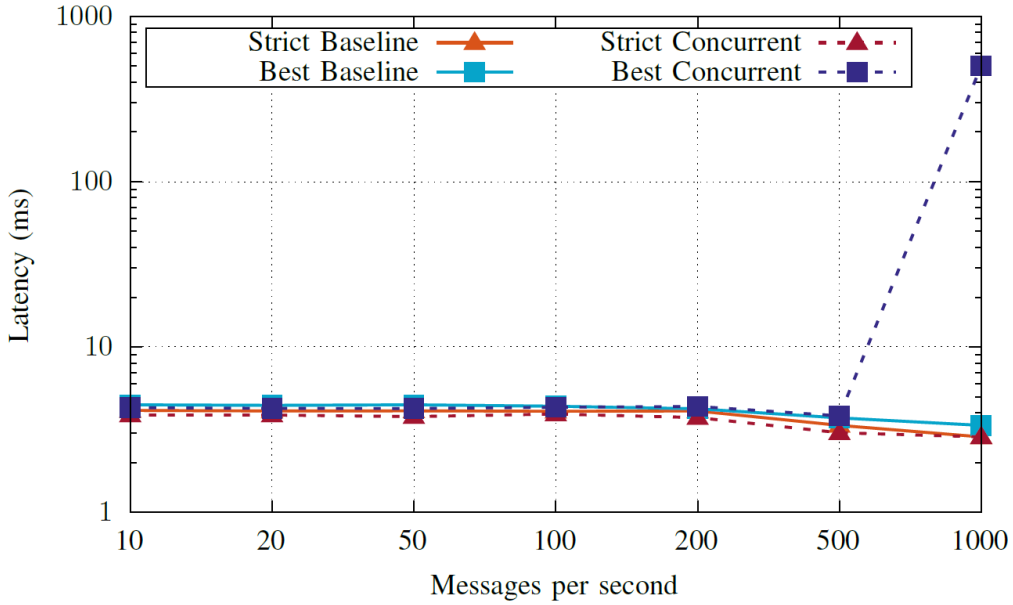

Figura 2.4: *Test della latenza end-to-end per le QoS supportate (BQ e SQ), prima eseguite separatamente e poi concorrentemente con numero di eventi generati da 0 a 1000 al secondo*

Capitolo 3

Ebpf

Contents

BPF, acronimo di *Berkeley Packet Filter*, è una tecnologia nata nel 1992 e utilizzata inizialmente per fare *packet filtering*. Una tecnica utilizzata per filtrare i pacchetti in arrivo o in uscita da una determinata interfaccia di rete. Un esempio classico di utilizzo potrebbe essere il *firewalling*. In questo caso solitamente viene analizzato l'*header* del pacchetto per ottenere informazioni, tra cui l'indirizzo sorgente, per poi decidere in base a un insieme di regole definite in precedenza, se accettare il pacchetto (PASS) o scartarlo (DROP). Tra le regole che si possono adottare, le più comuni sono quelle di utilizzare una *blacklist* o una *whitelist*, nel primo caso si ha una lista che rappresenta tutte le caratteristiche che un pacchetto deve avere per essere scartato (ad esempio se proviene da una determinata sorgente), nel secondo caso si ha invece una lista che rappresenta tutte le caratteristiche che un pacchetto deve avere per essere accettato. Chiaramente il secondo approccio seppur molto più limitativo permette una sicurezza maggiore. Se queste opzioni risultano troppo statiche per sistemi che hanno necessità di un alto livello di dinamicità, un altro metodo che si può adottare è il *dynamic packet filtering*. Quest'ultimo permette una gestione dinamica delle regole con cui gestire i pacchetti, ad esempio, chiudendo o aprendo le porte in ascolto in base alle necessità. Come accennato prima, il *packet*

filtering permette di analizzare gli *header* dei pacchetti, operando al livello tre del modello OSI, quindi a livello di rete, per questo motivo non è permesso accedere al *payload* del pacchetto per utilizzarlo come discriminante della decisione da effettuare. È chiaro che il *packet filtering* ha assunto un ruolo fondamentale sin dall'avvento di Internet, tuttavia, gli ideatori Steven McCanne e Van Jacobson hanno dimostrato come l'approccio introdotto da BPF sia 20 volte più veloce del *packet filtering* utilizzato all'epoca e considerato allo stato dell'arte. L'innovazione introdotta da BPF risiede principalmente nell'introduzione di due concetti, l'introduzione di una macchina virtuale (che lavora direttamente con i registri della CPU) e la capacità di operare e prendere decisioni senza dover analizzare l'intero pacchetto con conseguente utilizzo limitato della memoria.

Nel 2014 viene introdotta da Alexei Starovoitov la versione estesa di BPF (da qui il nome eBPF, *extended Berkeley Packet Filter*). Con l'obbiettivo di adattarsi all'evolversi dell'*hardware* i risultati ottenuti con questa estensione sono di notevole interesse, in particolare, eBPF risulta fino a 4 volte più veloce rispetto all'implementazione originale e con la possibilità di caricare programmi con una complessità maggiore grazie all'aumento dei numeri di registri utilizzabili (da due da 32 bit a dieci da 64 bit) [8].
Inizialmente l'utilizzo di BPF era limitato allo spazio *kernel* quando per la prima volta fu esteso anche allo spazio utente ne fu chiaro il potenziale e non venne più visto come limitato allo stack di rete ma divenne paragonabile a un modulo *kernel*.

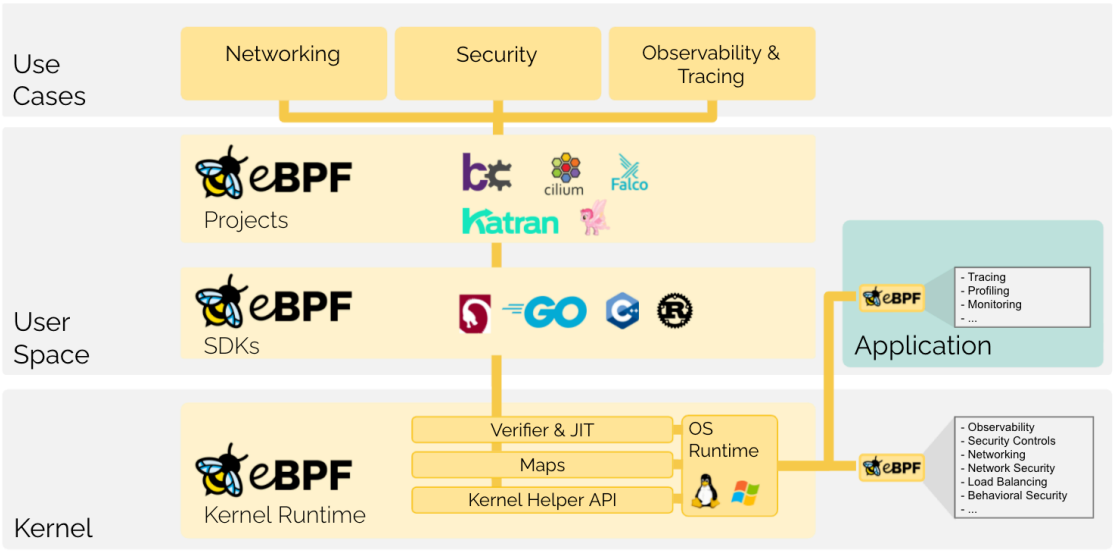

Figura 3.1: *Architettura eBPF*

In sintesi, eBPF è una tecnologia all'avanguardia che permette di estendere le capacità di un *kernel linux* senza dover ricorrere a modifiche onerose o invasive come la modifica del codice del *kernel* stesso, o il caricamento nello stesso di moduli separati (che richiederebbero di ricompilare l'intero *kernel*). Il principale vantaggio che porta l'utilizzo di eBPF è quello di poter operare a livello del *kernel* dove si ha una visione complessiva del sistema e conseguentemente con un controllo maggiore. Questo approccio risponde ai limiti riguardanti le difficoltà riscontrate nell'evoluzione del *kernel* causate dalla necessità di avere un codice estremamente sicuro e robusto. Permette inoltre, di ottenere sistemi con livelli di performance molto elevati su diversi aspetti come il *networking* (agendo, ad esempio, anzitempo sulla gestione dei pacchetti) o grazie ad aspetti di *load-balancing* piuttosto ché di *monitoring* del sistema. [5] Quello che nell'effettivo viene permesso di fare è l'esecuzione di una macchina virtuale astratta, che esegue codice in un ambiente isolato. Se si dovesse fare un paragone con un linguaggio di programmazione si potrebbe prendere in considerazione la JVM (*Java Virtual Machine*) come ambiente isolato, dove esegue programma *Java* in contrapposizione al programma BPF. Le istruzioni BPF vengono tradotte da linguaggi come il C o Rust grazie ad appositi compilatori, e infine eseguiti all'interno del *kernel*. Per rispondere ai requisiti di robustezza e sicurezza del codice iniettato nel *kernel* quest'ultimo passa attraverso una fase di testing. Come si può vedere in Figura 3.2, esiste un modulo chiamato *Verifier* attraverso il quale il programma BPF deve passare in modo da evitare codice che faccia saturare la memoria del *kernel*, crei cicli infiniti e/o che per qualsiasi altro motivo si blocchi, causando l'interruzione dell'intera macchina. Insieme al *Verifier* esiste un compilatore JIT (*Just-In-Time*) in grado di tradurre e compilare le istruzioni BPF.

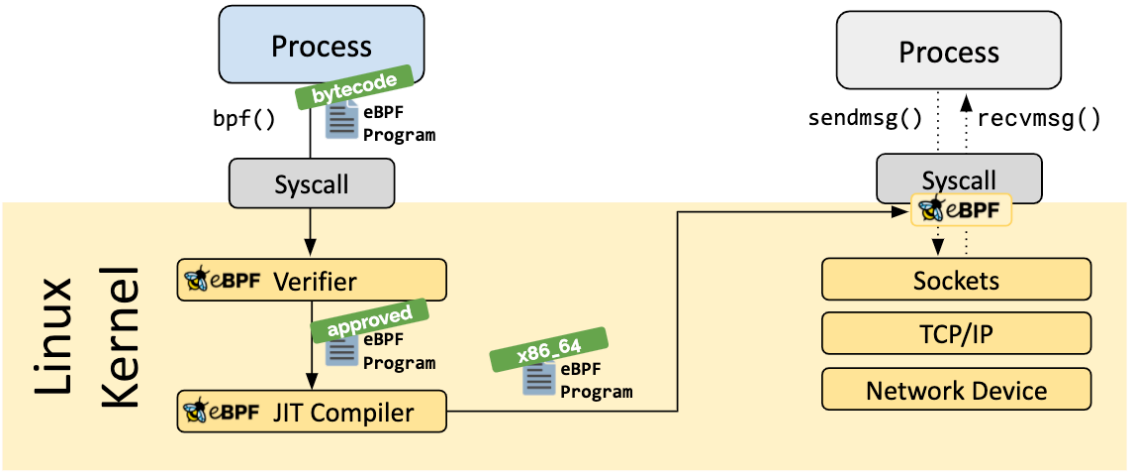

Figura 3.2: *Schema del processo di caricamento di un programma BPF*

Il *kernel* è un sistema basato su eventi (apertura di un file, inizializzazione di un processo, ecc.), un programma BPF permette di eseguire il codice compilato e

precedentemente iniettato all'avvenire di determinati eventi.

3.1 Architettura e componenti

Come accennato precedentemente diversi componenti sono necessari per poter eseguire un programma eBPF, i principali sono:

- Verifier
- Compilatore JIT
- Funzioni helper
- Chiamate tail
- Mappe

3.1.1 Verifier

Ricordando che un programma BPF esegue in un determinato punto del *kernel* al verificarsi di un determinato evento, risulta fondamentale che il codice iniettato ed eseguito sia idoneo ed efficiente. A questo proposito il componente chiamato in causa è il *Verifier*. Il codice può essere rifiutato per diversi motivi, tra cui, cicli incontrollati, numero massimo di istruzioni superato, presenza di codice irraggiungibile (che causerebbe il caricamento di codice inutile, determinando anche un ritardo nell'esecuzione del programma BPF) e in ultimo, e forse il più pericoloso, l'accesso a sezioni non permesse. Per quanto riguarda il limite massimo di istruzioni, all'introduzione di eBPF risultava essere di 4096 per programma (estendibile nel caso in cui si facciano programmi BPF innestati, anche qui con alcune limitazioni), e 32768 come limite di complessità, inteso come numero massimo di istruzioni esplorate dal *Verifier* durante i diversi percorsi di esecuzione (meno le istruzioni soggette a pruning dell'albero di ricerca). Al momento della stesura di questa tesi, tuttavia, il limite di complessità è stato aumentato considerevolmente fino a un milione. Dunque, per garantire un certo grado di affidabilità il *Verifier* opera nella seguente maniera, inizialmente un'analisi statica viene effettuata controllando che il programma abbia effettivamente una fine, questo controllo viene completato in seguito alla costruzione di un grafo aciclico diretto basato sul codice. Il grafo viene costruito con la seguente rappresentazione: ogni istruzione rappresenta un nodo e ogni nodo è collegato all'istruzione successiva. Infine, viene analizzato il grafo con una ricerca di tipo *depth-first* cercando potenziali percorsi ciclici. Il *pruning* dell'albero di ricerca avviene al momento dell'analisi di un nuovo ramo, guardando i precedenti stati raggiunti fino a quel punto, se uno di questi contiene lo stato corrente allora l'istruzione

da prendere in considerazione si può saltare (in quanto quel particolare stato era già stato accettato). Il controllo non avviene solamente sui registri ma anche sullo *stack*, le funzioni in cui questo controllo è effettuato sono rispettivamente *regsafe()* e *states_equal()* [7]. Al termine di questa fase il *Verifier* può garantire che il codice del programma BPF che si vuole caricare non presenta cicli ricorsivi e nessun percorso rischioso.
Conseguentemente vengono analizzate le singole istruzioni con l'obbiettivo di verificare che non ci siano istruzioni non valide e che i puntatori alla memoria siano correttamente dereferenziati. Infine, viene controllato che qualunque sia il percorso seguito dal programma attraverso le istruzioni, si arrivi sempre all'istruzione di terminazione del programma BPF. Il *Verifier* è sicuramente un software complesso, al momento della stesura di questa tesi il codice sorgente del *Verifier* è di circa 14 mila linee di codice (in Figura 3.3 se ne può notare la rapida crescita) e in quanto tale è difficile garantire la totale correttezza. Risulta infatti che talvolta programmi catalogati come sicuri in realtà non lo siano nella loro totalità e in contrapposizione programmi che invece lo sono, vengano rifiutati (basti pensare che i programmi troppo complessi da analizzare sono scartati e catalogati come non sicuri). Inoltre, con l'utilizzo di una *llvm*, un'infrastruttura di compilazione che permette di ottimizzare la fase di compilazione, l'elusione del *Verifier* risulta più semplificata. Questo deriva dalla difficoltà nello scrivere una porzione di codice che effettui un'analisi statica di un'altra porzione di codice.

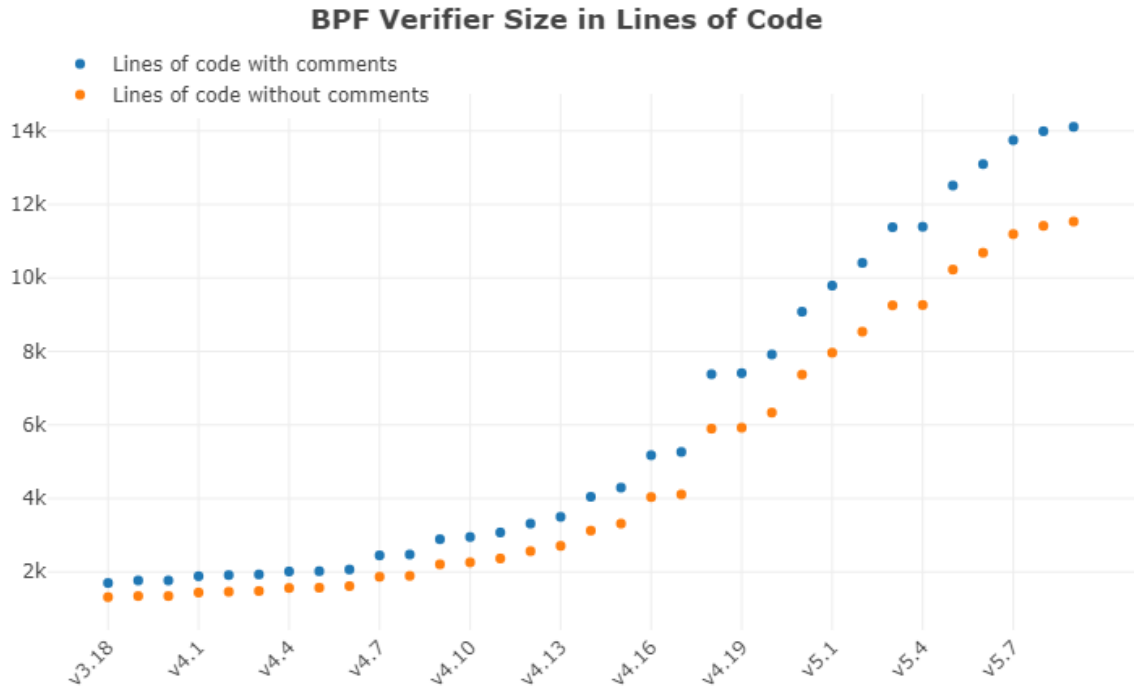

Figura 3.3: *Line di codice del Verifier per versione di Linux [1]*

3.1.2 Chiamate Tail

Il concetto delle chiamate *Tail* rappresenta la possibilità di chiamare programmi BPF da altri programmi BPF, permettendo di scomporre la complessità in sotto-programmi più semplici. Così come esiste un limite di istruzioni eseguibile esiste anche un limite di chiamate a programmi BPF innestati, 32. Uno dei vantaggi delle chiamate *Tail* è quello di poter superare il limite di istruzioni eseguibili imposte dal *Verifier*, impostandole e innestandole in maniera corretta. Tuttavia, se la semantica lo richiede, è importante prevedere un sistema di comunicazione tra i diversi programmi, in quanto di default i dati non sono condivisi tra programmi diversi. A questo proposito, e anche con l'obbiettivo di condividere dati con lo spazio utente, sono state introdotte le mappe BPF. Insieme a questa caratteristica, il fatto che le chiamate *Tail* siano unidirezionali le rende molto efficienti e paragonabili a un semplice *jump* nel codice. Chiaramente le chiamate *Tail* possono essere effettuate solo tra programmi BPF dello stesso tipo.

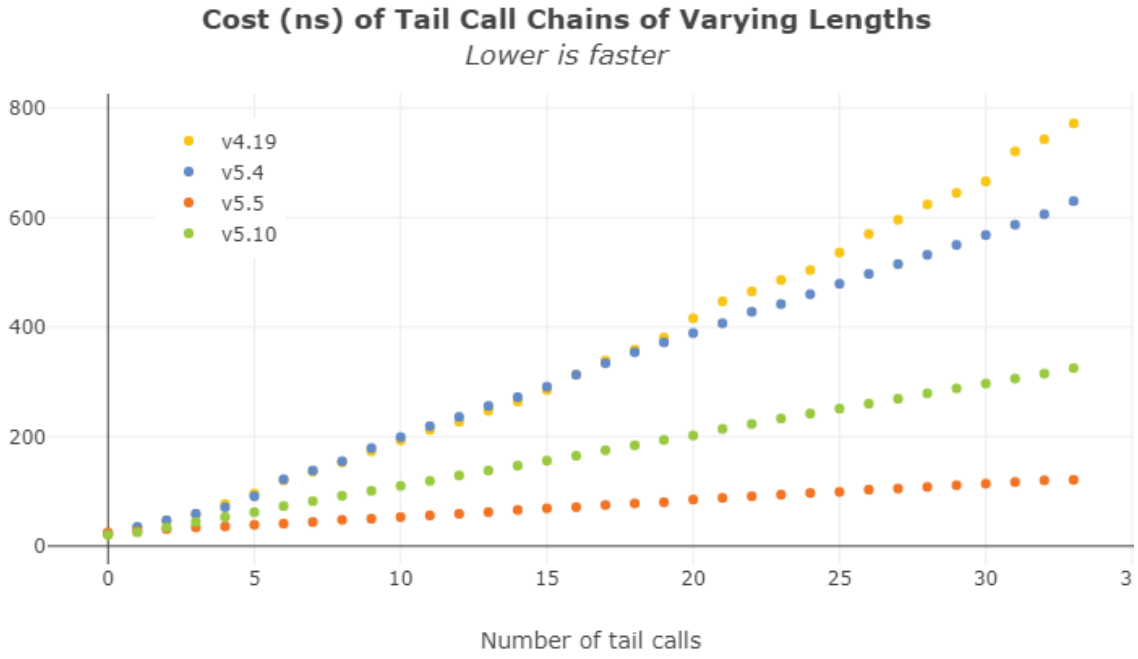

Figura 3.4: *Costo in ms delle chiamate Tail per versione di Linux [2]*

3.1.3 Mappe

Una mappa BPF, è una struttura dati di tipo chiave/valore visibile e accessibile sia da programmi BPF che da programmi nello spazio applicativo nello spazio utente. I dati che si possono inserire posso essere di diversa natura (il *kernel* considera chiavi e valori come *binary large object*, senza preoccuparsi del tipo di dato), purché sia

definita la loro dimensione a tempo di compilazione. In particolare, una mappa è definita da [6]:

- Tipo
- Numero massimo di elementi
- Dimensione in byte della chiave
- Dimensione in byte del valore

Da notare che le mappe sono limitate superiormente in quanto hanno numero massimo di elementi contenibili, questo chiaramente ne limita la scalabilità. L'accesso alla mappa da spazio utente avviene tramite un messaggio al *kernel*, a differenza dell'accesso dallo spazio *kernel* che può essere considerato come un'operazione atomica. Inoltre, nel caso in cui più programmi BPF abbiano visibilità e possibilità di accedere alla stessa mappa, possono verificarsi corse critiche risolvibili grazie ad alcuni lock introdotti da BPF. Le mappe sono dunque strutture dati utilizzate principalmente per condividere dati tra spazio kernel e spazio utente, ma anche, come accennato precedentemente, per condividere dati tra programmi BPF chiamati in una successione di chiamate *Tail*. Esistono diverse tipologie di mappe e una particolare tipologia permette di contenere al suo interno i descrittori di programmi BPF [8].

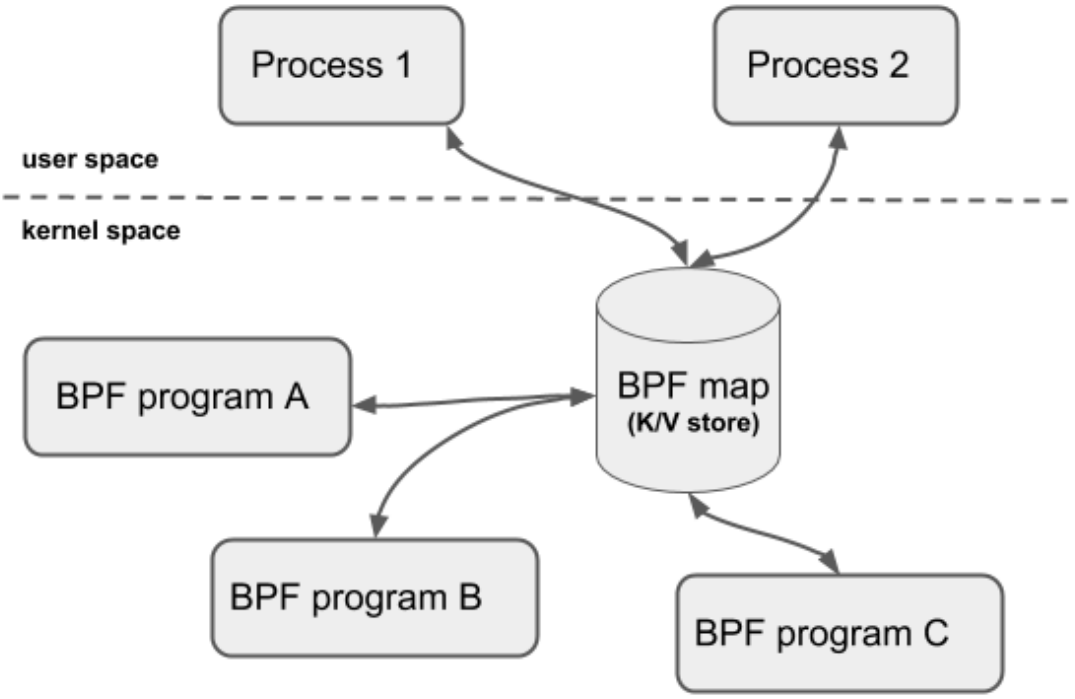

Figura 3.5: *Schema mappa BPF*

Alcune delle tipologie di mappe utilizzabili sono:

- **Hash Table Maps**: intesa inizialmente come mappa generica di BPF, sono simili alle classiche mappe basate su algoritmi di hashing e di conseguenza con tempo di accesso costante. Il tipo che le rappresenta è BPF_MAP_TYPE_HASH e la gestione dello spazio utilizzato dalla mappa è gestito dal *kernel*. Esiste anche la versione LRU (*least recently used*) dove, in caso la mappa sia piena, e ricordando che sono limitate superiormente, l'elemento che viene scartato è l'ultimo utilizzato in ordine di tempo.

- **Array Maps**: questa tipologia di mappa reincarna quello che è un *array*, gli elementi sono allocati precedentemente e impostati a 0, il valore chiave può essere solamente di 4 byte. Nonostante sia ottimizzata per effettuare la ricerca più veloce, questa tipologia di mappa presenta alcuni svantaggi, il primo è che l'operazione di update non è atomica quindi si possono avere inconsistenze nella lettura, il secondo è che gli elementi dell'array non possono essere cancellati e di conseguenza non è possibile ridurre la dimensione dell'array. Il tipo che le rappresenta è BPF_MAP_TYPE_ARRAY. Anche per questo tipo di mappa esiste la variante con politica LRU.

- **Program Array Maps**: sempre di tipo array ma in questo caso mantiene i descrittori di file rappresentanti i programmi BPF. Un uso classico di

questa tipologia di mappe è nel caso di utilizzo delle chiamate *Tail*. Si ricorda che il numero massimo di chiamate è di 32 e il tipo della mappa è BPF_MAP_TYPE_PROG_ARRAY.

- **Perf Event Array Maps**: specifiche nel mantenere eventi che il *kernel* vuole comunicare allo userspace. Si mette in attesa degli eventi generati dal *kernel*. Queste mappe sono utilizzate molto per effettuare *observability* e il tipo che le rappresenta è BPF_MAP_TYPE_PERF_EVENT_ARRAY.

- **Per-CPU Hash Maps**: come nel caso delle *Hash Table Maps*, ma ogni CPU ha visibilità della propria *Hash Map*. Utile nel caso si vogliano fare ricerche più efficienti e/o aggregare i vari risultati.

- **Per-CPU Array Maps**: come nel caso delle *Array Maps*, ma ogni CPU ha visibilità della propria *Array Map*. Utile nel caso si vogliano fare ricerche più efficienti e/o aggregare i vari risultati.

- **Cgropus Array Maps**: Questa tipologia mantiene riferimenti a dei *cgroups*, che sono sostanzialmente dei moduli del *kernel* che permettono di tenere controllato l'utilizzo delle risorse da parte di alcuni processi. Utile nel caso in cui si vuole che diversi programmi BPF accedano ai medesimi cgroups.

- **Device e CPU Map Maps**: le prime, definite da BPF_MAP_TYPE_DEVMAP, sono utilizzate nel campo del networking nel caso in cui si voglia gestire il traffico di rete. Contiene infatti riferimenti ai dispositivi di rete nel caso in cui si voglia, per esempio, eseguire una redirezione di un pacchetto. Le *CPU Map Maps* funzionano allo stesso modo, ma, invece che referenziare dispositivi di rete, contengono riferimenti alle varie CPU. I motivi per cui si potrebbe usare una mappa di questo genere sono: scalabilità e isolamento.

```c
int bpf_create_map(enum bpf_map_type map_type,
    unsigned int key_size,
    unsigned int value_size,
    unsigned int max_entries)
{
    union bpf_attr attr = {
        .map_type    = map_type,
        .key_size    = key_size,
        .value_size  = value_size,
        .max_entries = max_entries
    };

    return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));

}
```

Figura 3.6: *Funzione helper per la creazione di una mappa, restituisce il file descriptor della mappa creata*

```
enum bpf_map_type {
  BPF_MAP_TYPE_UNSPEC,  /* Reserve 0 as invalid map type */
  BPF_MAP_TYPE_HASH,
  BPF_MAP_TYPE_ARRAY,
  BPF_MAP_TYPE_PROG_ARRAY,
  BPF_MAP_TYPE_PERF_EVENT_ARRAY,
  BPF_MAP_TYPE_PERCPU_HASH,
  BPF_MAP_TYPE_PERCPU_ARRAY,
  BPF_MAP_TYPE_STACK_TRACE,
  BPF_MAP_TYPE_CGROUP_ARRAY,
  BPF_MAP_TYPE_LRU_HASH,
  BPF_MAP_TYPE_LRU_PERCPU_HASH,
  BPF_MAP_TYPE_LPM_TRIE,
  BPF_MAP_TYPE_ARRAY_OF_MAPS,
  BPF_MAP_TYPE_HASH_OF_MAPS,
  BPF_MAP_TYPE_DEVMAP,
  BPF_MAP_TYPE_SOCKMAP,
  BPF_MAP_TYPE_CPUMAP,
  BPF_MAP_TYPE_XSKMAP,
  BPF_MAP_TYPE_SOCKHASH,
  BPF_MAP_TYPE_CGROUP_STORAGE,
  BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
  BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
  BPF_MAP_TYPE_QUEUE,
  BPF_MAP_TYPE_STACK,
  /* See /usr/include/linux/bpf.h for the full list. */
};
```

Figura 3.7: *Alcuni tipi di mappe rappresentate da un enumerativo*

Esistono tuttavia altre tipologie di mappe non ancora citate come le *Socket Maps*, anche qui in versione *Hash Map* e *Array Map* e le informazioni contenute sono i riferimenti agli elementi necessari per aprire una connessione attraverso una socket, i tipi sono rispettivamente BPF_MAP_TYPE_SOCKMAP e BPF_MAP_TYPE_SOCKHASH. Le ultime tipologie di mappe a cui si vuole fare cenno sono caratterizzate dalla politica di accesso ad esse, tra queste troviamo: *Queue Map*, con politica di accesso FIFO (*first in first out*), *Stack Map*, con politica di accesso LIFO (*last in first out*). Nel caso delle *Queue Maps* quando si cerca un elemento il risultato sarà dato dall'elemento che più è rimasto nella mappa, in contrapposizione alle Stack Maps dove l'elemento restituito è l'ultimo ad essere entrato.

Per quanto riguarda l'accesso concorrente, essendo le mappe strutture dati condivise tra diversi programmi, il rischio di accesso contemporaneo con conseguente verificarsi di corse critiche è reale. A tal proposito è stata introdotta la possibilità di utilizzare dei lock. In questo caso prima di accedere alla mappa, generalmente per

modificare un elemento, si applica un *lock* con la funzione helper *bpf_spin_lock*, si eseguono le operazioni e al termine si rilascia il *lock* con la funzione *bpf_spin_unlock*. Il *lock* per ovvi motivi di efficienza si applica al singolo elemento acceduto e non all'intera mappa. Nell'effettivo questo meccanismo opera come un semaforo. Il comportamento desiderato è quello di un'operazione atomica, ottenuto con un possibile degrado delle performance a causa dell'introduzione dei *lock*. Essendo però programmi che richiedono alta efficienza è bene capire quando è necessario introdurre questo meccanismo. I *lock*, tuttavia, non sono utilizzabili per tutte le tipologie di mappe ma solo per le *Array Maps*, *Hash Maps* e *Cgroup Maps*.

3.1.4 Compilatore JIT

I classici compilatori durante il corso della storia hanno presentato limiti importanti, come la difficile portabilità del codice compilato (a causa delle diverse infrastrutture hardware presenti), la poca efficienza durante l'esecuzione della compilazione, ecc.. Inoltre, si sono dimostrati più adatti a linguaggi di programmazione caratterizzati da una tipizzazione statica, dove il tipo di dato è definito in maniera statica a tempo di compilazione e che non cambia durante l'intero processo di esecuzione. Nascono così gli interpreti, in grado di eseguire un programma senza doverlo prima compilare, ma con il grande svantaggio della poca efficienza nell'esecuzione del codice se paragonato alla versione compilata. Un metodo che risulta migliore rispetto a un interprete puro, in termini di efficienza, può essere caratterizzato dalle macchine virtuali (la *Java Virtual Machine* ne è un esempio), tuttavia, le prestazioni non sono ancora nemmeno simili a una versione di codice interamente compilata e a questo punto entrano in gioco i compilatori JIT, con un tentativo di ottenere i benefici di entrambi. In breve, un compilatore JIT è capace di compilare il codice a run-time (invece che prima dell'esecuzione) e generalmente consiste nella trasformazione del *bytecode* in codice macchina. È in grado di analizzare il codice e individuare quali sono le parti di codice migliori da sottoporre a previa compilazione, con conseguente trasformazione in codice macchina; dunque, per il proseguo dell'esecuzione del programma sarà il codice compilato a essere chiamato in causa. Un compilatore JIT, tuttavia, deve trovare un compromesso, è importante individuare e decidere quali sono le parti da compilare e quando, in questo caso si riescono ad ottenere buone prestazioni. In eBPF il compilatore JIT inizia la compilazione non appena il *Verifier* termina il processo di verifica di sicurezza del codice, oltre a tradurre e compilare le istruzioni BPF il compilatore JIT è in grado di tradurre in linguaggio macchina il *bytecode* BPF. Lo stato della memoria durante l'esecuzione di un compilatore JIT risulta frammentato, in particolare in alcune parti della memoria si hanno regioni eseguibili mentre altre sono scrivibile, dunque, un meccanismo di protezione potrebbe essere necessario.

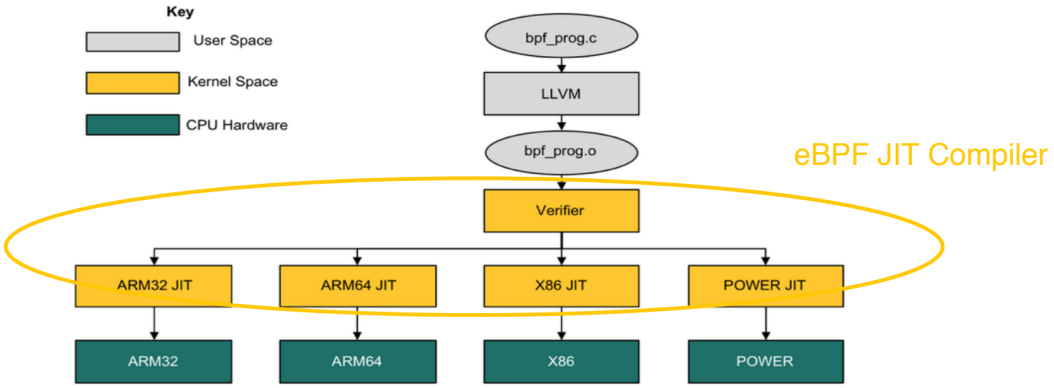

Figura 3.8: *Schema con collocazione del compilatore JIT in eBPF [10]*

3.1.5 Funzioni helper

Infine, le funzioni helper rappresentano le classiche funzioni che possono essere utili in qualsiasi tipo di programma, come la generazione di numeri random, l'accesso alla data e all'orario corrente, funzioni per la lettura, scrittura o ricerca per le mappe e così via. Una delle funzioni helper più importanti è sicuramente quella che permette il caricamento di un programma BPF nel kernel (load_bpf_file).

3.2 Networking e XDP

Con il passare del tempo il networking ha assunto un ruolo sempre più importante, in particolare con l'avvento dei sistemi distribuiti su cloud, è aumentata considerevolmente la necessità di scambio di dati tra server appartenenti al cloud o a semplici datacenter. Con l'aumentare del traffico aumenta conseguentemente anche la necessità di processare i pacchetti in entrata o in uscita. Come già accennato, eBPF può essere utilizzato per diverse cose, tra cui observability, security, networking, tracing e profiling, in questa sezione si approfondirà l'aspetto del networking e in particolare di XDP. Per le sue caratteristiche, come l'efficienza e la programmabilità, eBPF risulta la tecnologia ideale per il packet filtering, aggiungendo funzionalità al kernel. Il controllo del traffico eseguito dal kernel permette di operare in diversi modi, dalla possibilità di scartare o prioritizzare i pacchetti in base al tipo alla distribuzione del carico. Tuttavia, a volte può essere necessario operare sui pacchetti ad uno step precedente, per questo l'utilizzo di programmi XDP può diventare la

soluzione da adottare. Un programma XDP, infatti, è eseguito appena dopo che il pacchetto arrivi al network driver ma prima che entri nello stack della rete, potendo operare così, su una rappresentazione del pacchetto senza i metadati. Il vantaggio principale si ha sull'efficienza ma con lo svantaggio di non poter usufruire dei metadati e delle strutture dati fornite dal kernel.

XDP sta per Express Data Path e si può definire come un processore di pacchetti, sicuro, performante, programmabile e integrato nel kernel che esegue programmi BPF quando un driver NIC (Network Interface Card) riceve un pacchetto, prendendo decisioni su come agire sui pacchetti stessi [LIBRO]. Da notare che l'aggettivo performante sia dovuto al fatto che con questa configurazione non sia possibile anticipare nel tempo l'analisi dei pacchetti più di quanto XDP faccia già. Tuttavia, l'efficienza e le alte prestazioni non sono dovute al solo punto di esecuzione ma anche ad altri fattori come la non allocazione della memoria durante il processamento di pacchetti con XDP o il non accesso ai metadati del pacchetto o più semplicemente perché per definizione e per costruzione un programma eBPF deve avere una terminazione rapida. Un'altra importante caratteristica è che XDP permette di analizzare i pacchetti senza dover pagare l'overhead dello stack TCP/IP. In sostanza, con un programma XDP si può decidere di eseguire determinate azioni sui pacchetti anche in base al tipo. Modifica, ritrasmissione, cancellazione di un pacchetto sono alcune delle opzioni permesse e i codici risultanti di XDP sono: [https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html]:

- **XDP_DROP**: molto utilizzato nel caso di utilizzo di XDP per casi di firewalling, o di mitigazione di attacchi DDoS (Distributed Denieal of Service), dove il pacchetto per diversi motivi, come la malformazione del pacchetto stesso, la non affidabilità della sorgente del pacchetto, ecc. viene scartato e di conseguenza non arriva allo stack di rete del kernel. L'utilizzo della CPU e di energia in questo caso è minimo e questo contribuisce ad essere un ottimo candidato se si vuole agire per prevenire attacchi di tipo DDoS.
- **XDP_PASS**: uno dei casi più frequenti, in questo caso il programma XDP permette il passaggio allo stack di rete del kernel del pacchetto in ingresso. Questa decisione può essere, ad esempio, effettuata in seguito al controllo del contenuto del pacchetto è in seguito alla verifica che la sorgente non faccia parte di una blacklist o che faccia parte di una whitelist. Da notare che prima del passaggio allo stack di rete del kernel il contenuto del pacchetto può essere modificato.
- **XDP_TX**: in questo caso il pacchetto subisce un forwarding, ovvero dopo essere stato letto e/o modificato viene rimandato sulla NIC (Network Interface Card) su cui è arrivato. Utilizzabile nel caso in cui si voglia avere un load

balancer di tipo one-legged, non idoneo quindi nel caso di un multi-NIC load-balancer.

- **XDP_REDIRECT**: il comportamento in questo caso è molto simile al caso precedente ma qui avviene una redirezione del pacchetto verso una nuova NIC. Si può utilizzare per un load-balancer di tipo multi-NIC.

- **XDP_ABORTED**: questo codice viene ritornato da un programma XDP nel caso di errore e non dovrebbe essere ritornato come valore di ritorno di un programma funzionante. Per questo assume sempre il valore 0, chiaramente con il risultato che il pacchetto viene scartato.

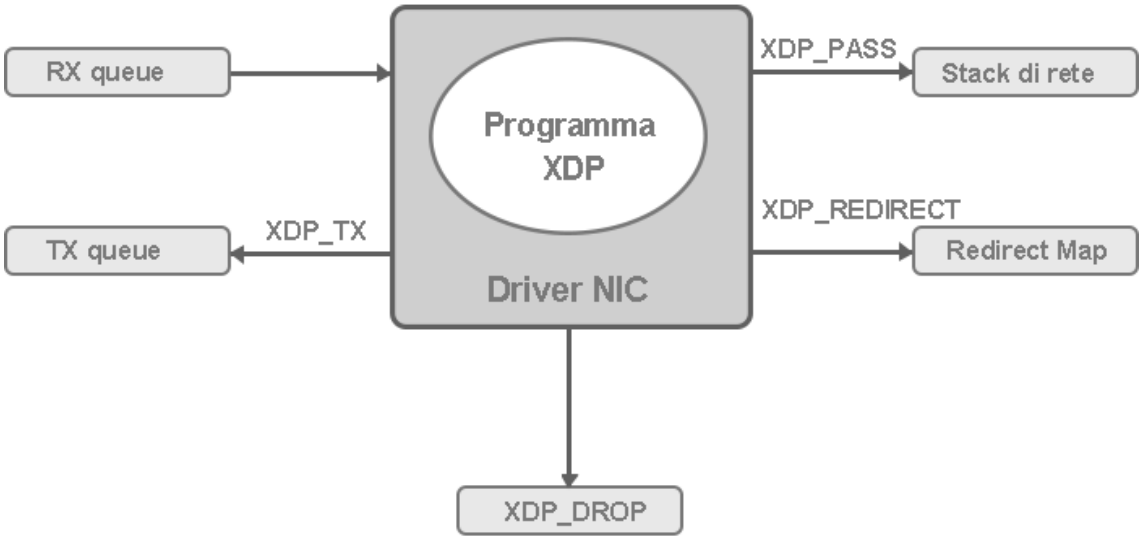

Figura 3.9: *Azioni possibili in un programma XDP*

```
enum xdp_action {
    XDP_ABORTED = 0,
    XDP_DROP,
    XDP_PASS,
    XDP_TX,
    XDP_REDIRECT,
    };
}
```

Figura 3.10: *Codici di ritorno contenuti in linux/bpf.h*

XDP non accedendo al network stack del kernel non ha accesso alla struttura sk_buff (a differenza, ad esempio, di tc), una struttura dati che contiene tutte le informazioni di controllo per il pacchetto in arrivo, ma utilizza una particolare struttura chiamata xdp_buff (una rappresentazione del pacchetto ma senza i metadati). Xdp_buff è utilizzata, tra le altre cose, a presentare il context sul quale il programma XDP potrà lavorare. Il vantaggio sicuramente sta nell'efficienza mentre lo svantaggio principale è quello di non poter utilizzare i metadati per una migliore rappresentazione del pacchetto (il motivo è semplicemente dovuto al fatto che i metadati sono contenuti ad un livello più alto della pipeline della ret).

3.2.1 Metodi di esecuzione

XDP generalmente fa riferimento a un'interfaccia di rete e ogni qualvolta che un pacchetto arriva il programma XDP viene eseguito tramite una callback. Inoltre, i driver che supportano XDP, per motivi di efficienza, possono al più contenere un programma per volta. Nel caso in cui si voglia avere una catena di programmi non è possibile farlo caricandone più di uno ma si può ottenere questo risultato utilizzando il costrutto relativo alle chiamate tail prima descritto. XDP supporta tre diversi metodi di esecuzione, ognuno eseguito in un punto diverso e la principale differenza nelle tre possibilità sta nell'efficienza. Le tre modalità di esecuzione sono:

- **Generic XDP**: in questo caso il programma è caricato all'interno del kernel, fornisce le peggiori prestazioni ed è vista come una modalità utilizzata laddove non vi è supporto diretto dell'hardware per l'utilizzo di xdp. Ottimo nel caso in cui si voglia testare il corretto funzionamento di un programma XDP. In linux utilizzando il comando "ip" si può procedere al caricamento di un oggetto BPF compilato in una delle tre modalità, per il caso generic si puo usare il

seguente comando:

```
# ip link set dev enps03 xdpgeneric obj xdp.o
```

Dove xdp.o è l'oggetto BPF compilato e enps03 è il nome del dispositivo di rete.

- **Native XDP**: questo è considerato il caso di default dei modelli di esecuzione di un programma XDP, in questa metodologia il programma richiede il supporto del driver della scheda di rete in quanto è lì che verrà effettivamente caricato il programma. Con questa modalità le azioni sui pacchetti vengono eseguiti il prima possibile (a livello software), si vedrà, nel caso successivo, che è possibile fare ancora meglio ma con supporto dell'hardware. In questo caso le prestazioni sono migliorate rispetto alla modalità Generic XDP. Il comando per caricare un programma in questa modalità è:

```
# ip link set dev enps03 xdpdrv obj xdp.o
```

- **Offloaded XDP**: quest'ultimo caso è forse quello più interessante in quanto è la modalità di esecuzione di XDP più performante. Nel caso offloaded, infatti, è permesso di scaricare l'intero programma sull'hardware. Lo svantaggio in questo caso è che è necessario disporre di una SmartNIC, una scheda di rete programmabile. Il guadagno si ha sulla non necessità di eseguire il programma sulla CPU ma sulla scheda di rete ottenendo prestazioni molto elevate. Un altro svantaggio rispetto alle due precedenti modalità è che non tutte le funzioni helper che fornisce bpf sono utilizzabili così come nnon tutte le tipologie di mappe sono supportate. Il comando per utilizzare questa modalità è:

```
# ip link set dev enps03 xdpoffload obj xdp.o
```

Come anticipato prima non è possibile caricare più programmi XDP alla volta e quindi il comando restituisce un messaggio di errore se si prova ad effettuare il caricamento di un programma su un'interfaccia che già ne contiene uno, nel caso in cui si voglia sovrascrivere il programma precedente si può utilizzare "-force" come opzione per il comando "ip". Inoltre, nel caso in cui non si specifichi quale modalità utilizzare per il caricamento del programma XDP, come nel seguente esempio:

```
# ip link set dev enps03 xdp obj xdp.o
```

verrà fatto un tentativo per il caricamento di default, ovvero in modalità native, nel caso in cui questa non sia supportata, verrà utilizzata la modalità generic

[https://docs.cilium.io/en/latest/bpf/]. Per rimuovere un programma da un'interfaccia si può utilizzare il comando

```
# ip link set dev enps03 xdp off
```

Chiaramente non è necessario indicare il nome del programma in quanto il supporto al caricamento, come anticipato in precedenza, è limitato ad un programma per volta.

Capitolo 4

Realizzazione di un MoM per funzioni di Routing Acceleration

Contents

In questo capitolo si analizzerà nel dettaglio il lavoro svolto in questa tesi per l'estensione effettuata sul *Message-oriented Middleware*, componente principale della sezione *Delivery* di TEMPOS (introdotta nella Sezione 2.1.2). L'obbiettivo è quello di realizzare un modello *Publish/Subscribe* accelerando la fase di routing in seguito all'arrivo di un messaggio sul *Message-oriented Middleware*, redirigendolo su un altro nodo e permettendo l'esecuzione del routing con minore interruzione. Il modello *Publish/Subscribe* è una forma di comunicazione asincrona utilizzata, tra le altre cose, in contesto *Serverless*. Tra le diverse applicazioni, più di interesse per questa tesi, si vede infatti la propagazione degli eventi a diverse destinazioni. Il modello *Publish/Subscribe* è solitamente composto da:

- *Publisher*
- *Pub/Sub Message Broker*

- *Subscriber*

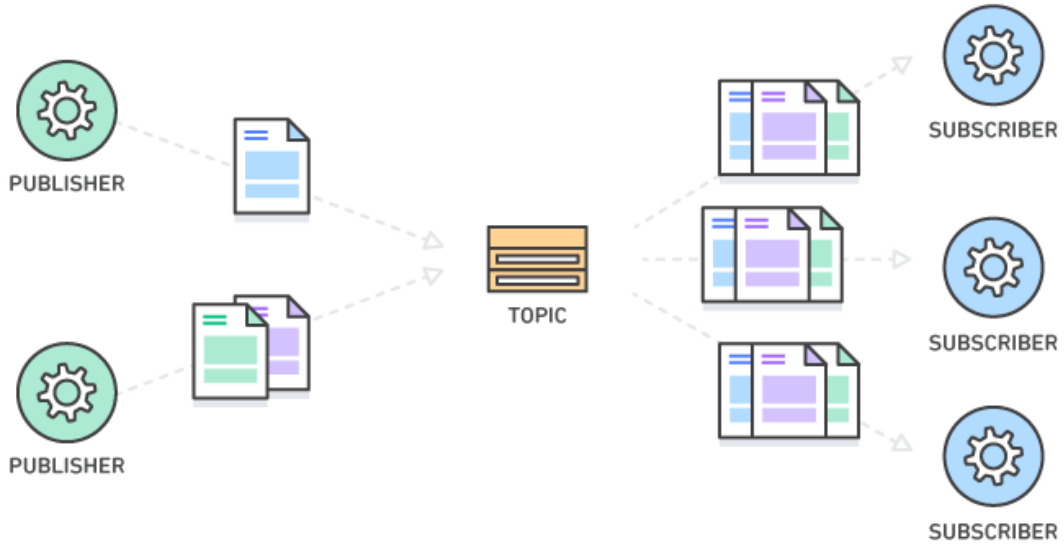

Figura 4.1: *Schema modello Publish/Subscribe [12]*

La comunicazione non è *one-to-one*, infatti l'invio di un messaggio da parte del *Publisher* avviene senza un destinatario preciso, ma può, ad esempio, essere categorizzato in maniera tale che il *Message Broker* possa poi inoltrarlo solo ai *Subscriber* interessati alla suddetta categoria. Nel caso di TEMPOS, una delle possibili opzioni vede il *Publisher* rappresentato dai *Trigger*, il *Message Broker* dal *MoM* (oggetto di questa tesi) mentre i *Subscriber* dagli *Executor Node*. Per quanto riguarda il discriminante per la categoria si tratta di un intero di 8 bit, all'inizio del pacchetto, che rappresenta il *Topic* di interesse per i *Subscriber*. Da notare che il messaggio è consumato da un solo *Subscriber* scelto nella lista di *Subscriber* iscritti a quel particolare *Topic*. Il processo di esecuzione, dunque, dovrà essere:

1. Arrivo di un pacchetto UDP sul *Message-oriented Middleware*
2. Analisi del *Topic* dell'evento da inoltrare
3. Individuazione dei *Subscriber* iscritti a quel *Topic*
4. Inoltro a uno dei *Subscriber* individuati

In Figura 4.2 si può vedere l'esempio di un possibile *workflow* dello schema appena citato, in questo caso i *Trigger* fanno da *Publisher* mentre gli *Executor* fanno da *Subscriber*. Da notare che questo non è l'unico comportamento possibile.

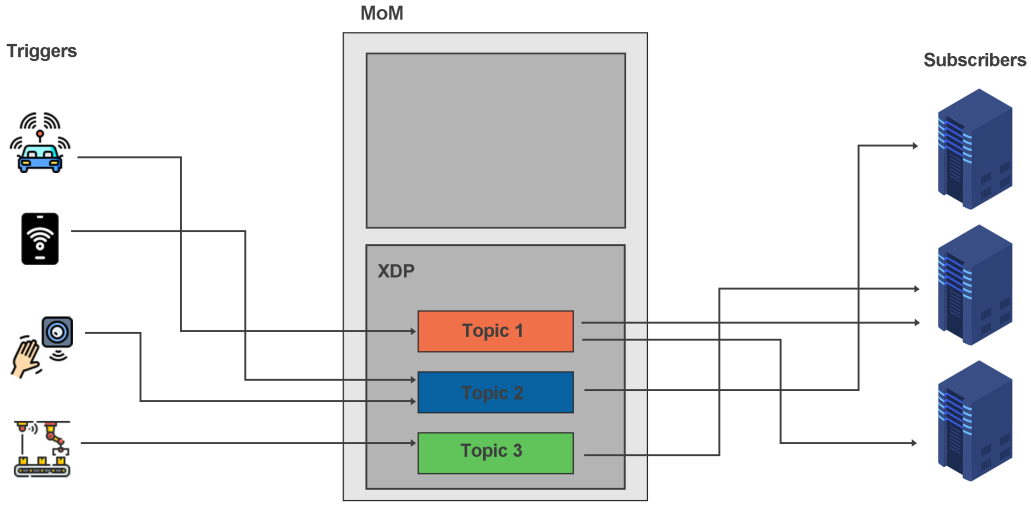

Figura 4.2: *Esempio di workflow in cui i Publisher sono i Trigger mentre i Subscriber sono gli Executor*

4.1 Progettazione

Per rispettare tempi di esecuzione ridotti, specialmente per casi in cui sia richiesta una *Quality of Service* di tipo *Strict*, si è deciso di utilizzare eBPF come tecnologia per la realizzazione di questa tesi, in particolare del modulo XDP (discusso nel Capitolo 3). Grazie a questa tecnologia è possibile estendere le funzionalità del *kernel*, permettendo l'esecuzione di programmi prima dello stack TCP/IP del *kernel* Linux. In questo modo l'arrivo di un pacchetto, che deve essere rediretto verso i nodi *Subscriber*, innesca l'esecuzione di un programma appositamente scritto per la gestione dello stesso. La particolarità di questo approccio risiede nella capacità di eseguire il programma senza uscire dal *kernel space*. In questo modo si evita l'overhead che inevitabilmente si accumulerebbe in seguito alla copia del pacchetto a livello applicativo (nello *user space*), evitando *malloc*, copia di byte in memoria per la gestione del pacchetto ecc, ritenute operazioni costose anche se gestite dal sistema operativo. Si ricorda che grazie a XDP le operazioni effettuate sui pacchetti sono efficienti e sono eseguite il prima possibile subito dopo l'arrivo dello stesso sul *driver NIC* [8].

4.1.1 Architettura e componenti

Tra i componenti principali sicuramente troviamo il programma XDP, che dovrà essere opportunamente caricato sull'interfaccia di rete di interesse, a tal proposito verrà scritto un programma che eseguirà nello *user space* e che prima di effettuare il caricamento dovrà ricevere in ingresso l'indice rappresentante l'interfaccia di rete. È necessario inoltre avere una base di dati che specifichi quali *Subscriber* siano interessati a quali *Topic*. Tuttavia, non è possibile utilizzare i classici metodi di condivisione dei dati, non sarebbe possibile infatti accedervi da un programma XDP, in quanto esso agisce al di fuori dello *user space*. A tal proposito si è deciso di usare una mappa BPF (Sezione 3.1.3), una speciale struttura dati che offre la possibilità di essere condivisa anche da programmi che risiedono nel *kernel space*.

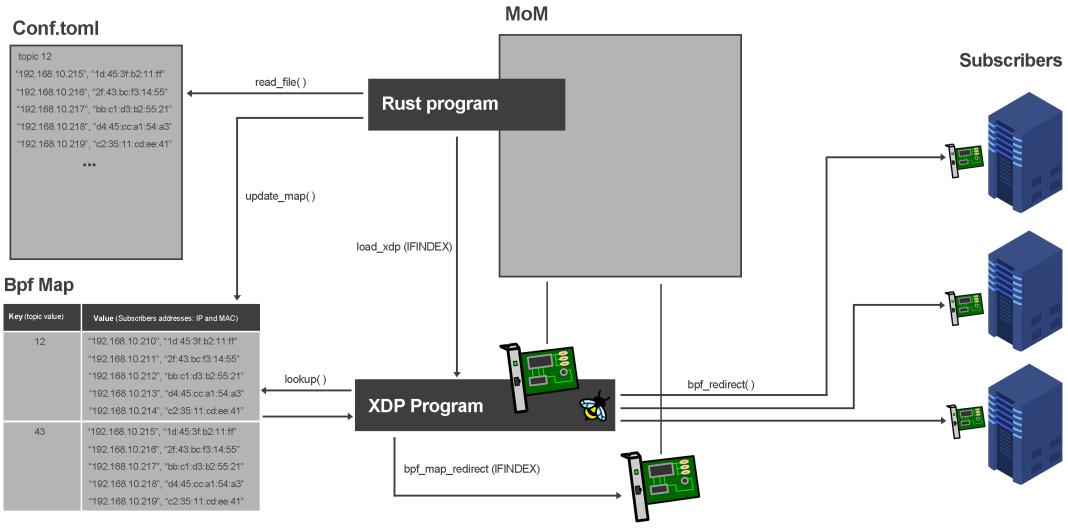

Figura 4.3: *Schema dell'architettura*

4.1.2 Fasi della progettazione

Come anticipato prima, la fase iniziale che innesca l'inizio del processo di esecuzione è l'arrivo di un pacchetto UDP, dal quale bisogna prima estrarre il valore del *Topic*. Il pacchetto verrà anticipato da un programma XDP, gestito e instradato verso il nuovo nodo.

Prima fase: Lettura valore del *Topic*

Per arrivare alla lettura del valore del Topic bisogna prima introdurre come XDP funziona. Innanzitutto, XDP lavora solo in condizioni di pacchetti non frammentati e con informazioni sui puntatori ad inizio e fine pacchetto. Non si ha accesso completo ai metadati, rappresentati normalmente dalla struttura *sk_buff*, ma se ne ha una rappresentazione più leggera chiamata *xdp_buff*. Un programma XDP riceve in ingresso un puntatore a una struttura che rappresenta il contesto, l'equivalente BPF di una struttura *xdp_buff*, contenente oltre ad alcuni metadati, i puntatori ad inizio e fine pacchetto in arrivo. A questo punto è utile introdurre velocemente la composizione di un pacchetto UDP e l'incapsulamento che subisce il pacchetto per arrivare alla destinazione seguendo il protocollo *TCP/IP*, che è caratterizzato da questi layer:

1. **Network Access Layer**: aggiunge un *header* contenente informazioni del pacchetto relative al layer fisico, ad esempio e di interesse per questa tesi, gli indirizzi MAC

2. **Internet Layer**: ha il compito di controllare il flusso e il routing del traffico, l'*header* contiene informazioni come gli indirizzi ip di sorgente e destinazione

3. **Transport Layer**: complice dell'effettivo trasporto dei dati, ad esempio, attraverso il protocollo UDP

4. **Application Layer**: il livello più alto, generalmente per le interazioni con gli applicativi

Stando a quanto appena descritto, per accedere al valore del *Topic*, contenuto nel primo byte del *payload* di un pacchetto UDP, si utilizzerà un puntatore che sia in grado di passare oltre tutti gli *header* e accedere direttamente all'indirizzo di memoria contenente l'informazione cercata. Gli *header* in sequenza saranno dunque il *Frame Header*, aggiunto dal *Network Layer*, seguito dall'*IP Header* aggiunto dall'*Internet Layer* ed infine l'*UDP header* relativo al *Transport Layer*.

Seconda fase: Individuazione dei *Subscriber*

Una volta letto il valore del *Topic* assegnato al pacchetto bisogna essere in grado di determinare l'assegnazione uno a molti relativa al valore letto e ai *Subscriber* associati. È importante che l'informazione relativa alla lista dei *Subscriber* associati al *Topic* sia reperibile a runtime sia dallo *user space* che dal *kernel space*, in modo tale che possa essere aggiornata, eventualmente dallo *user space*, e acceduta a runtime dal programma XDP in esecuzione. Tra le mappe messe a disposizione da eBPF verrà utilizzata una *Hash-Table Map* dove la chiave sarà rappresentata dal valore del *Topic*,

e il valore sarà l'insieme delle informazioni necessarie per instradare il pacchetto ai *Subscriber* iscritti a quel *Topic*. Queste informazioni saranno inizialmente contenute in un file di configurazione, opportunamente scritto, e lette da un programma che esegue nello *user space* che si occupa anche di creare e popolare la mappa sopracitata. Per la scelta sul *Subscriber* al quale instradare il pacchetto si utilizzerà una politica di tipo FIFO (*First In First Out*).

Terza fase: Modifica del pacchetto

A questo punto dell'esecuzione si hanno informazioni relative al *Topic* e alla lista dei *Subscriber* associati, con indirizzo Ip e indirizzo MAC di ognuno di essi. Analizzando il ruolo del *Message-oriented Middleware* l'obiettivo è quello di inoltrare il pacchetto al *Subscriber* scelto secondo la politica FIFO dalla lista dei *Subscriber* associati. Il primo step sarà un controllo generale sul pacchetto, sugli *header*, sul protocollo utilizzato, sulla porta di ingresso ecc., allo scopo di determinare la provenienza e le decisioni da effettuare per la gestione dello stesso. In seguito, si inizieranno a sovrascrivere gli indirizzi di memoria che fanno riferimento all'instradamento del pacchetto in ingresso. In particolare, nell'*header* relativo al *Network Access Layer* si hanno informazioni sugli indirizzi fisici di sorgente e destinazione, questi dovranno essere sovrascritti. Nel header introdotto dall'*Internet Layer*, invece, troviamo gli indirizzi ip di sorgente e destinazione, che anch'essi dovranno essere sovrascritti con i dati relativi al *Subscriber*. Infine, dopo alcuni controlli e aggiornamenti dei campi di integrità del pacchetto, si può procedere all'invocazione delle funzioni *helper* per l'instradamento del pacchetto verso il nuovo nodo. Sarà importante effettuare tutte le operazioni rispettando regole ben precise introdotte dal *Verifier* (Sezione 3.1.1), affinché il programma in *bytecode* possa essere validato, compilato dal *JIT Compiler* (Sezione 3.1.4) e infine caricato.

4.2 Implementazione

In questa sezione si spiegheranno e si discuteranno le scelte implementative, discutendone vantaggi e svantaggi ove possibile.

4.2.1 Limiti

Ci sono da tenere in considerazione importanti limiti richiesti da un programma eBPF come, ad esempio, la non possibilità di allocare memoria dinamica. Durante l'esecuzione di un programma normalmente si ha la possibilità di accedere alla memoria tramite lo *stack* e il *heap*. In breve, lo *stack* rappresenta la memoria statica a cui si può fare riferimento, si dice statica perché la sua dimensione è nota a tempo di compilazione, rendendolo efficiente ma utilizzabile solo per le strutture

dati con dimensione fissa e nota a priori. Per quanto riguarda il *heap* rappresenta la memoria allocata dinamicamente, può aumentare lo spazio allocato anche a tempo di esecuzione e di conseguenza può contenere strutture dati la quale dimensione non sia nota a priori. In un programma BPF si ha accesso solo alla memoria relativa allo *stack*, sicuramente un approccio limitativo ma che permette maggiore sicurezza ed efficienza. In Rust, ad esempio, non è possibile utilizzare tutto ciò che fornisce la *standard library* perché alcune funzioni necessitano di allocazioni nello *heap*, di conseguenza si usa la direttiva:

```
1 #![no_std]
```

4.2.2 Librerie e componenti

Per avere una visione generale, ampia e semplificata si può riassumere il sistema come composto da tre componenti principali:

- programma XDP: che effettivamente processa i pacchetti in ingresso con il compito di eseguire il routing verso altri nodi
- programma in user space: con il compito di caricare nel *kernel* il programma xdp e di popolare opportunamente la mappa BPF
- mappa BPF: struttura dati in condivisione tra *user space* e *kernel space* a cui si interfacciano i programmi

Per motivi anche di efficienza il linguaggio utilizzato per la realizzazione del *middleware* TEMPOS è Rust; dunque, per lavorare con eBPF e questo linguaggio si sono individuate due possibili librerie:

- **libbpf-rs**: un *wrapper* rust della libreria *libbpf* scritta in C. Garantisce tutte le caratteristiche di BPF ma il programma in se deve essere scritto in c con estensione ".bpf.c"
- **aya**: libreria sperimentale che consente di scrivere anche i programmi BPF in Rust ma che non copre tutte le caratteristiche fornite da BPF

Sono state inizialmente esplorate entrambe le soluzioni separatamente concludendo però utilizzando la libreria *libbpf-rs*, in quanto forniva l'implementazione di tutte le caratteristiche di BPF; tuttavia, per completezza verranno mostrate entrambe le implementazioni, commentandole opportunamente. Sarà inoltre necessario introdurre delle astrazioni per poter lavorare con le varie strutture dati che rappresentano i vari *header* dei pacchetti. Nel caso di utilizzo della libreria *Aya* si potrà usufruire dei binding generati dalla libreria stessa, di seguito un esempio di astrazione in Rust della struttura contenente i dati relativi agli indirizzi ip.

```rust
pub type __u8 = ::aya_bpf::cty::c_uchar;
pub type __u16 = ::aya_bpf::cty::c_ushort;
pub type __u32 = ::aya_bpf::cty::c_uint;
pub type __be16 = __u16;
pub type __be32 = __u32;

#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct iphdr {
    pub _bitfield_align_1: [u8; 0],
    pub _bitfield_1: __BindgenBitfieldUnit<[u8; 1usize]>,
    pub tos: __u8,
    pub tot_len: __be16,
    pub id: __be16,
    pub frag_off: __be16,
    pub ttl: __u8,
    pub protocol: __u8,
    pub check: __sum16,
    pub saddr: __be32,
    pub daddr: __be32,
}
```

Listing 4.1: *Definizione struttura iphdr relativa all'Internet Layer*

Utilizzando la libreria *libbpf-rs* il codice del programma BPF sarà scritto in C e sarà sufficiente includere i file *header* forniti da Linux contenenti le meta informazioni.

```c
#include <linux/if_ether.h>
#include <linux/in.h>
#include <linux/ip.h>
#include <linux/udp.h>
```

Listing 4.2: *File di inclusione per la gestione dei pacchetti*

```c
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
  __u8  ihl:4,
    version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
  __u8  version:4,
      ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
  __u8  tos;
  __be16  tot_len;
  __be16  id;
  __be16  frag_off;
  __u8  ttl;
```

```
16   __u8  protocol;
17   __sum16 check;
18   __be32  saddr;
19   __be32  daddr;
20   /*The options start here. */
21 };
```

Listing 4.3: *Definizione della struttura iphdr nel file ip.h*

Di interesse per questa tesi, i campi più importanti saranno gli indirizzi di destinazione e di sorgente, id del pacchetto e i dati relativi alla consistenza del pacchetto.

Un altro importante componente sarà una mappa necessaria per mantenere le informazioni relative ai *Subscriber* associati ad ogni *Topic*. Tra le mappe messe a disposizione si è deciso di utilizzare una *Hash-Table Map*, il cui tipo risulta essere *BPF_MAP_TYPE_HASH*. Essendo una mappa che si basa su algoritmi di tipo hash, il tempo per l'accesso alla chiave cercata sarà sempre costante, rendendola un'ottima candidata per contenere anche grandi quantità di coppie chiave-valore. Una caratteristica importante considerando la quantità di pacchetti in ingresso per unità di tempo e che la ricerca avviene per ognuno di essi. Non sarà necessario dunque cambiare tipologia di struttura nel caso in cui il numero di *Topic* dovesse aumentare considerevolmente. Tuttavia, le mappe in BPF rappresentano per definizione problemi riguardanti la scalabilità, in quanto, il numero di entry che possono al massimo contenere è definito a tempo di compilazione. Nel caso del *Message-oriented Middleware* e dell'implementazione attuale si è deciso di utilizzare un numero massimo di *entry* pari a 256, in modo da poter mappare tutti i valori che possono essere rappresentati dal *Topic* (si ricorda essere un intero a 8 bit all'inizio del *payload* di un pacchetto). Per quanto riguarda il valore di chiave sarà un singolo byte a rappresentare il *Topic*, mentre per il valore la gestione è leggermente più complessa. Essendo un dato che deve avere definizione fissa a tempo di compilazione, si utilizzerà come dimensione un valore superiore al numero dei *Subscriber* associati al *Topic* con il maggior numero di *Subscriber* (5 nel caso dell'implementazione corrente). In questo modo sarà necessario aggiungere un *padding* nel file di configurazione o di generarlo nel programma che si occupa della deserializzazione del file. Per le informazioni relative ad ogni *Subscriber* all'interno del valore della mappa si sono utilizzati 10 byte, 4 byte per la rappresentazione dell'indirizzo ip e 6 byte per la rappresentazione dell'indirizzo MAC.

```
1 #define ADDRESSES_PER_TOPIC 5
2 struct bpf_map_def SEC("maps") subs_addresses = {
3   .type = BPF_MAP_TYPE_HASH,
4   .key_size = sizeof(unsigned char),
5   .value_size = sizeof(char) * 10 * ADDRESSES_PER_TOPIC, //4 (ip) + 6(
     mac) * 5 (possible destinations)
```

```
  .max_entries = 256,
};

struct address_meta {
    __u32 ip[ADDRESSES_PER_TOPIC];
    unsigned char mac[ADDRESSES_PER_TOPIC][6];
};
```

Listing 4.4: *Definizione della mappa e della struttura sulla quale verrà mappato il valore dell'entry relativa al Topic del messaggio*

Per garantire disaccoppiamento tra dati e codice si è deciso di utilizzare un file di configurazione, che dovrà contenere esattamente le stesse informazioni in modo da poter mappare precisamente tutti i byte nelle *entry* della *Hash-Table Map*. Il file sarà letto attraverso un programma Rust che risiede nello *user space* e che attraverso l'utilizzo di uno *Skeleton*, generato automaticamente dalla libreria *libbpf-rs*, sarà in grado di aggiornare la mappa in condivisione con il programma XDP. Per evitare dipendenze varie si è deciso di utilizzare un file di configurazione di tipo *TOML*, lo stesso utilizzato per la gestione delle dipendenze del programma. Mentre per la lettura e la deserializzazione di sono utilizzate delle strutture di appoggio apposite. Il codice può essere ottimizzato ma si è deciso di mantenerlo il più comprensibile possibile, in quanto essendo eseguito solamente all'inizializzazione del sistema non necessita elevata efficienza.

```
[[topic]]
topic_value = 1

    [[topic.address]]
    ip="10.0.0.211"
    mac="40:a6:b7:66:38:fb"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
```

```
    mac="0:0:0:0:0:0"

[[topic]]
topic_value = 2

    [[topic.address]]
    ip="10.0.0.212"
    mac="40:a6:b7:66:39:ba"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"

    [[topic.address]]
    ip="0.0.0.0"
    mac="0:0:0:0:0:0"
```

Listing 4.5: *porzione di file di configurazione TOML contenente le informazioni relative ai Subscriber associati ai Topic*

```
#[derive(Deserialize, Debug, Clone)]
struct Address {
    ip: String,
    mac: String,
}

#[derive(Deserialize, Debug, Clone)]
struct Topic {
    topic_value: u8,
    address: Vec<Address>,
}

#[derive(Deserialize, Debug, Clone)]
struct Config {
    topic: Vec<Topic>
}
```

Listing 4.6: *Definizione strutture utilizzate per il mapping con il file di configurazione*

Prima di mostrare l'inserimento dei dati all'interno della mappa è necessario generare lo skeleton utilizzando come input il file ".bpf.c".

```rust
use std::fs::create_dir_all;
use std::path::Path;
use libbpf_cargo::SkeletonBuilder;

const SRC: &str = "./src/bpf/xdpredirect.bpf.c";

fn main() {
    create_dir_all("./src/bpf/.output").unwrap();
    let skel = Path::new("./src/bpf/.output/xdpredirect.skel.rs");
    SkeletonBuilder::new(SRC).generate(&skel).unwrap();
    println!("cargo:rerun-if-changed={}", SRC);
}
```

Listing 4.7: *Programma per la generazione dello skeleton basato sul programma XDP*

Una volta generato il file *Skeleton* si potrà accedere alle mappe definite dal programma XDP.

```rust

let mut file = File::open(PATH_FOR_CONFIG)
    .expect("Error: opening config file");
let mut data = String::new();
file.read_to_string(&mut data)
    .expect("Error: reading config file.");
let topics: Config = from_str(&data).unwrap();
for topic in topics.topic.into_iter(){
    skel.maps_mut().subs_addresses()
    .update(&[topic.topic_value],
        &hashmap_key_per_topic(&topic.address),
        MapFlags::NO_EXIST)?;
}
```

Listing 4.8: *Lettura file e inserimento dei dati nella mappa*

```rust
fn hashmap_key_per_topic(addresses: &Vec<Address>) -> [u8; VALUE_SIZE]
    {
    //first insert all ip addresses then all mac addresses
    let mut res: [u8; VALUE_SIZE] = [0; VALUE_SIZE];
    let mut index = 0;
    for address in addresses.into_iter(){
        // Adding all ip addresses
        let octets = ip_string_to_array(&address.ip);
        for octet in octets{
            res[index] = octet;
            index+=1;
        }
    }
    for address in addresses.into_iter(){
        // Adding all mac addresses
```

```
        let bytes = mac_string_to_array(&address.mac);
        for byte in bytes{
            res[index] = byte;
            index+=1;
        }
    }
    return res;
}

fn ip_string_to_array(ip: &String) -> [u8; 4]{
    let ip_addr = Ipv4Addr::from_str(&ip);
    let ip_addr = match ip_addr {
        Ok(ip_addr) => ip_addr.octets(),
        Err(_) => panic!("File formatting error")
    };
    return ip_addr;
}

fn mac_string_to_array(mac: &String) -> [u8; 6]{
    let mac_addr = MacAddress::from_str(&mac);
    let mac_addr = match mac_addr {
        Ok(mac_addr) => mac_addr.bytes(),
        Err(_) => panic!("File formatting error")
    };
    return mac_addr;
}
```

Listing 4.9: *Funzione che esegue il parsing da un vettore contenente le stringhe di indirizzi ip e MAC*

Per terminare il caricamento del programma XDP avviene sempre tramite lo *Skeleton* precedentemente generato.

```
let opts = Command::from_args();
let skel_builder = XdpredirectSkelBuilder::default();
let open_skel = skel_builder.open()?;
let mut skel = open_skel.load()?;
let link = skel.progs_mut().xdp_redirect().attach_xdp(opts.ifindex)?;
skel.links = XdpredirectLinks {
    xdp_redirect: Some(link),
};
```

Listing 4.10: *Caricamento del programma XDP sull'interfaccia definita negli argomenti*

4.2.3 Programma XDP

Un programma XDP fa da processore di pacchetti ogni qualvolta che un pacchetto arriva sul *driver NIC*. Inizialmente bisogna dichiarare il punto di ingresso per

il programma e il nome *ELF* della sezione (in questo caso "xdp"). Per prima cosa sarà necessario analizzare il pacchetto, capire se è integro, che protocollo utilizza, la porta, e così via. Nel caso del *middleware* in questione verranno rediretti solamente i pacchetti che rispondono a determinati requisiti, facendo i controlli su:

- Internet protocol: solamente pacchetti che utilizzano il protocollo IPV4
- Frammentazione: non sono ammessi pacchetti frammentati
- Transport protocol: solo pacchetti UDP
- Porta: solo pacchetti che arrivano sulla porta 9998

Inoltre si noti che la rappresentazione dei dati nei pacchetti sia diversa, è necessario convertire i dati utilizzando delle funzioni apposite in grado di convertire i dati dalla rappresentazione *big-endian* ad una di tipo *little-endian*.

```
SEC("xdp")
int xdp_redirect(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    int pkt_sz = data_end - data;
    struct ethhdr *eth;
    struct iphdr *iph;
    struct udphdr *udp;
    __u16 pkt_size;
    __u8 protocol;
    __u32 off = sizeof(struct ethhdr);
    unsigned char *payload;
    unsigned char topic_value;

    eth = data; // Pointer to the start of ethernet layer header (containing mac addresses)
    iph = data + off; // Pointer to the start of network layer header (containing ip addresses)

    if (iph + 1 > data_end)
        return XDP_DROP;

    // Only ipv4 packets are proccessed
    if (eth->h_proto != bpf_htons(ETH_P_IP)){
        return XDP_PASS;
    }

    // Fragmented packets are not supported
    if (iph->frag_off & IP_FRAGMENTED)
        return XDP_PASS;

```

```
30      // Only UDP packets are proccessed
31      if (iph->protocol != IPPROTO_UDP) {
32          return XDP_PASS;
33      }
34
35      // Pointer to the start of transport layer header (UDP)
36      udp = (void *)iph + sizeof(*iph);
37      if ((void *)udp + sizeof(*udp) > data_end)
38          return XDP_PASS;
39
40      //Check port
41      if (udp->dest != bpf_ntohs(PORT))
42          return XDP_PASS;
43  ...
```

Listing 4.11: *Inizio del metodo per il processamento dei pacchetti, inizializzazione strutture e controllo dati del pacchetto*

Si noti che affinché il programma venga accettato dal *Verifier* sia necessario controllare che i puntatori a cui si accede siano all'interno dei dati del pacchetto, in modo da non accedere a memoria non consentita. Se questo controllo non venisse effettuato il *Verifier* rifiuterebbe il caricamento del programma generando dei messaggi di errore. Una volta effettuati tutti i controlli se il pacchetto non ha subito il passaggio al *network stack* vuol dire che deve essere soggetto a redirezione, si può quindi procedere alla lettura del primo byte che rappresenterà il valore del *Topic*.

```
1   ...
2       unsigned int payload_size;
3       payload_size = bpf_ntohs(udp->len) - sizeof(*udp);
4       //Pointer to the start of the payload
5       payload = (unsigned char *)udp + sizeof(*udp);
6       if ((void *) payload + payload_size > data_end)
7           return XDP_PASS;
8
9       if (payload + 1 > data_end)
10          return XDP_PASS;
11
12      // payload has at least one byte of data, first byte will be
        considered the topic value
13      topic_value = payload[0];
14  ...
```

Listing 4.12: *Controllo sul payload e lettura del Topic*

Conoscendo il valore del *Topic* ora si può utilizzare una *helper function* per accedere alla mappa utilizzando il valore appena letto come chiave. In seguito a quest'operazione si avranno gli indirizzi di tutti i *Subscriber* opportunamente deserializzati e mappati in una struttura dati.

```
struct address_meta {
    __u32 ip[ADDRESSES_PER_TOPIC];
    unsigned char mac[ADDRESSES_PER_TOPIC][6];
};
```

Listing 4.13: *Struttura dati contenente gli indirizzi di tutti i Subscriber associati ad un Topic*

```
...
    struct address_meta *addr;
    addr = bpf_map_lookup_elem(&subs_addresses, &topic);
...
```

Listing 4.14: *Lettura del valore nella mappa associato al Topic*

Per la decisione del *Subscriber* al quale inoltrare il pacchetto, come detto in precedenza, si vuole utilizzare una politica FIFO. Tra le opzioni valutate per tenere traccia dell'ultimo *Subscriber* a cui è stato inoltrato un pacchetto ci sono:

- opzione 1: mantenere il valore nella mappa dei *Subscriber* associati
- opzione 2: mantenere il valore in un array all'interno del programma XDP

La prima opzione sicuramente garantisce un maggior disaccoppiamento tra codice e dati, ma con lo svantaggio di dover sostenere un update sulla mappa ad ogni pacchetto in ingresso. La seconda opzione invece garantisce prestazioni più elevate ma non garantisce lo stesso livello di disaccoppiamento. Si è comunque scelto di procedere con la seconda opzione in quanto è chiaramente più importante ottenere meno *overhead* possibile durante la fase di routing.
A livello di implementazione si è creato un *array* di una lunghezza pari al numero di *Topic* che quest'ultimo riesce a mappare, dunque 256. Il valore ad ogni indice rappresenta l'ultimo *Subscriber* a cui è stato inoltrato il pacchetto. Si noti che per avere valori della mappa sempre costanti (requisito delle mappe BPF), laddove vi sono meno *Subscriber* rispetto al valore massimo di *Subscriber* possibili, si è utilizzato come indirizzo ip la stringa "0.0.0.0", così come per l'indirizzo MAC, rendendo necessario un controllo aggiuntivo. A questo punto, dopo aver controllato che l'*entry* sia stata trovata per superare i controlli del *Verifier*, si può procedere con la sovrascrittura della memoria contenente gli indirizzi di destinazione e sorgente relativi a indirizzo ip e MAC.

```
...
    if(addr) {
        // Index between (0 and ADDRESSES_PER_TOPIC) to pick the ip and
    mac addresses
        unsigned int next_available = addr_indexes[topic_value];

        if(next_available >= ADDRESSES_PER_TOPIC){
```

```
            return XDP_PASS;
        }

        if(addr[0].ip[next_available] == 0) { //Padding address
            next_available = 0;
        }

        memcpy(eth->h_source, eth->h_dest, sizeof(eth->h_source));
        memcpy(eth->h_dest, addr[0].mac[next_available], sizeof(eth->
    h_dest));

        __be32 tmp = iph->daddr;
        iph->daddr =  addr[0].ip[next_available];
        iph->saddr = tmp;
...
```

Listing 4.15: *Modifica del pacchetto sulla base dei dati letti dalla mappa*

Una volta modificati i dati relativi agli indirizzi bisogna rispristinare la consistenza del pacchetto, sovrascrivendo i vari *checksum* e l'id. Infine si può procedere con l'effettiva redirezione attraverso la funzione *helper bpf_redirect()*.

```
...
        // Id updating to avoid packet missing
        iph->id = iph->id + 1;

        // Checksum updating
        iph->check = 0;
        iph->check = ip_checksum((__u16 *)iph, sizeof(struct iphdr));

        if(next_available < ADDRESSES_PER_TOPIC)
            addr_indexes[topic_value] = ++next_available;
        else
            addr_indexes[topic_value] = 0;

        //Can use bpf_redirect_map() to support multiple devices
        return bpf_redirect(IFINDEX, 0);
    }

    //Entry not found in map
    return XDP_PASS;

}

static __always_inline __u16 ip_checksum(unsigned short *buf, int bufsz
    ) {
    unsigned long sum = 0;

    while (bufsz > 1) {
```

```
27          sum += *buf;
28          buf++;
29          bufsz -= 2;
30      }
31
32      if (bufsz == 1) {
33          sum += *(unsigned char *)buf;
34      }
35
36      sum = (sum & 0xffff) + (sum >> 16);
37      sum = (sum & 0xffff) + (sum >> 16);
38
39      return ~sum;
40  }
```

Listing 4.16: *Ripristino checkum e redirezione pacchetto*

Nel caso in cui si volesse estendere il programma per utilizzare diverse interfacce di rete, una soluzione possibile potrebbe essere quella di utilizzare la funzione *helper bpf_redirect_map()*, passando come argomenti una mappa (contenente l'indice dei dispositivi) e la chiave relativa al dispositivo che si vuole utilizzare. Questa opzione, tuttavia, è supportata solo in modalità di esecuzione native di XDP, quindi con il supporto del driver.

```
1  struct bpf_map_def SEC("maps") devs = {
2    .type = BPF_MAP_TYPE_DEVMAP,
3    .key_size = sizeof(int),
4    .value_size = sizeof(int),
5    .max_entries = 256,
6  };
7
8  ...
9  bpf_redirect_map(&devs, 1, 0);
10 ...
```

Listing 4.17: *Esempio con redirezione basata su una mappa di dispositivi*

In questo esempio il pacchetto viene rediretto al dispositivo associato all'indice 1 della mappa *devs*.

4.2.4 Compilazione e caricamento del programma XDP

Esistono diversi modi per caricare il programma XDP e inizializzare il sistema a fare routing, per questa tesi sono stati testati due diversi approcci. Il primo, più diretto, consiste nell'utilizzare il programma Rust scritto appositamente; dunque, sarà necessario eseguire la build del progetto attraverso il comando:

```
1  # cargo build --release
```

Si noti che *–release* sia necessario per avere una versione compilata ottimizzata. A questo punto si ha l'eseguibile pronto per eseguire il caricamento del programma:

```
# sudo ./target/release/xdp 2
```

Se il programma dovesse superare i controlli del *Verifier*, risulterà caricato e la mappa inizializzata con i dati deserializzati dal file di configurazione. In questo caso bisogna passare come argomento l'indice dell'interfaccia di rete sulla quale si vuole caricare il programma, e fino a quando il programma in *user space* non verrà terminato esso rimarrà sull'interfaccia indicata ad eseguire processamento di pacchetti. In questa modalità verrà eseguito il caricamento in modalità *native*, se questo non sarà possibile a causa della mancanza di disponibilità del driver, allora verrà caricato in modalità *generic*.

Un secondo approccio testato è quello di utilizzare iproute2 come loader del programma. In questo caso bisogna compilare il programma con estensione ".bpf.c" utilizzando il comando:

```
# cargo libbpf build
```

A questo punto si avrà nella cartella "/target/bpf" la versione compilata in bytecode del programma XDP (con estensione ".bpf.o"), pronta per passare attraverso il *Verifier* e conseguentemente attraverso il *JIT Compiler*. Per fare ciò, si utilizzerà *iproute2* attraverso il comando:

```
# sudo ip link set dev enp5s0 xdpgeneric obj xdpredirect.bpf.o sec
  xdp
```

oppure nel caso si volesse caricare in *native* mode:

```
# sudo ip link set dev enp5s0 xdpdrv obj xdpredirect.bpf.o sec xdp
```

Per verificare che il programma sia stato caricato correttamente è sufficiente verificarlo attraverso il comando:

```
# ip link
```

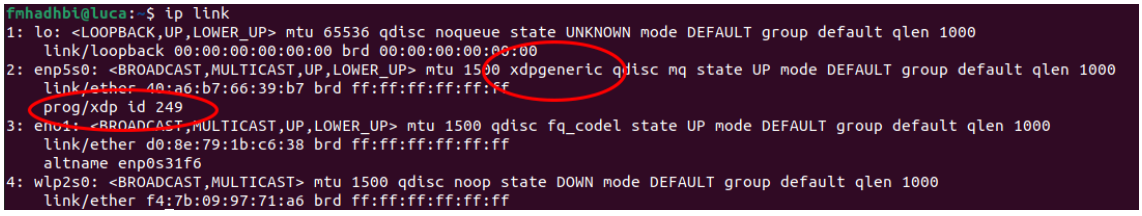

Figura 4.4: *Comando per la verifica del caricamento del programma XDP sull'interfaccia*

Verificato che il programma sia stato caricato correttamente rimane da effettuare il popolamento della mappa. Per questo si può utilizzare la libreria *bpftool*, in particolare, attraverso il comando:

```
# sudo bpftool map show
```

```
fmhadhbi@luca:~/tempos-final/target/bpf$ sudo bpftool map show
109: hash  name subs_addresses  flags 0x0
        key 1B  value 50B  max_entries 256  memlock 16384B
111: array  flags 0x0
        key 4B  value 32B  max_entries 1  memlock 4096B
112: array  name pid_iter.rodata  flags 0x480
        key 4B  value 4B  max_entries 1  memlock 4096B
        btf_id 281  frozen
        pids bpftool(21189)
113: array  flags 0x0
        key 4B  value 32B  max_entries 1  memlock 4096B
```

Figura 4.5: *Comando per visualizzare le mappe attualmente esistenti*

si può avere una visione di tutte le mappe. La mappa con id 109 risulta quella associata ai *Subscriber* e per aggiornare o aggiungere entry si può usare il comando:

```
# sudo bpftool map update id 109 key hex 01 value hex 0a 00 00 d3 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 a6 b7 66 38 fb 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00
```

A questo punto, per effettuare il *dump* della mappa e verificare che le entry siano corrette si può utilizzare la stessa libreria attraverso il comando:

```
# sudo bpftool map dump name subs_addresses
```

```
fmhadhbi@luca:~/tempos-final/target/bpf$ sudo bpftool map dump name subs_addresses
key:
01
value:
0a 00 00 d3 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 40 a6 b7 66  38 fb 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00
Found 1 element
```

Figura 4.6: *Comando per eseguire il dump di una mappa dato il nome*

Dopo aver inserito tutte le entry il programma XDP è pronto ad eseguire il routing. Dunque tutti i pacchetti che arriveranno sull'interfaccia e che supereranno i controlli, saranno intercettati dal programma XDP e rediretti senza che vengano passati allo *stack* di rete.

4.2.5 Sperimentazione con la libreria Aya e Rebpf

In questa sezione verrà mostrato il primo approccio utilizzato con l'obbiettivo di avere un programma interamente scritto in Rust. Anche se si sono comunque ottenuti alcuni risultati, tra cui la modifica dei dati del pacchetto, questa strada

è stata abbandonata per testare un approccio più consolidato (*libbpf-rs*). Dovendo infatti utilizzare più librerie *Rust* a causa della non copertura di tutte le funzionalità BPF, si andrebbe incontro a diverse definizioni per gli stessi costrutti. Oltre a ciò a causa di alcuni problemi relativi al binding per le varie librerie e per dipendere da meno librerie esterne si è preferito utilizzare un'unica libreria che racchiudesse in sé tutte le funzionalità di BPF, seppur dovendo scrivere il programma BPF in linguaggio C. Anche in questo caso è stato necessario scrivere due programmi differenti, uno con l'obbiettivo di eseguire all'interno del *kernel* e il secondo con l'obbiettivo di caricare il programma ed eventualmente gestirne le dipendenze. Trascurando il secondo, che seppur leggermente diverso non aggiunge nulla di significativo rispetto al caso introdotto per l'implementazione di questa tesi, si vuole introdurre il codice del programma relativo al *kernel* per le parti più importanti.

```
#[map(name = "NEW_DEST")]
static mut NEW_DEST: HashMap<u8, Addresses> = HashMap::<u8, Addresses
    >::with_max_entries(1024, 0);

#[map(name = "DEV_MAP")]
static mut DEV_MAP: BpfMapDef<u32, u32> = BpfMapDef::new(BpfMapType::
    DEVMAP, 1024);
```

Listing 4.18: *Definizione delle mappe utilizzate, relative a indirizzi di destinazione e dispositivi*

Per l'accesso agli indirizzi di memoria del pacchetto con gli opportuni controlli si è utilizzato il seguente metodo.

```
#[inline(always)]
unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*mut T,
    ()> {
    let start = ctx.data();
    let end = ctx.data_end();
    let len = mem::size_of::<T>();

    if start + offset + len > end {
        return Err(());
    }

    Ok((start + offset) as *mut T)
}
```

Listing 4.19: *Metodo per l'accesso puntuale agli indirizzi di memoria*

Si è reso necessario utilizzare Rebpf (un wrapper della libreria libbpf), oltre ad Aya, per poter chiamare la helper function.

```
const ETH_P_IP: u16 = 0x0800;
const ETH_HDR_LEN: usize = mem::size_of::<ethhdr>();

```

```
#[xdp]
pub fn xdp_prog(ctx: XdpContext) -> u32 {

    match try_xdp_prog(ctx) {
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}

fn get_map_value(key: u8) -> Option<&'static Addresses> {
    unsafe { NEW_DEST.get(&key) }
}

fn try_xdp_prog(ctx: XdpContext) -> Result<u32, ()> {

    let h_proto = u16::from_be(unsafe { *ptr_at(&ctx, offset_of!(ethhdr
    , h_proto))? });
    if h_proto != ETH_P_IP {
        return Ok(xdp_action::XDP_PASS);
    }
    //Check fragmented packets (?)

    let topic = u8::from_be(unsafe {*ptr_at(&ctx, 0)?});
    let mut action = xdp_action::XDP_PASS;

    //override destination if found in map
    match get_map_value(topic) {
        Some(addr) => {
            let start = ctx.data();
            let end = ctx.data_end();
            let len = mem::size_of::<u32>();
            let dest_offset = ETH_HDR_LEN + offset_of!(iphdr, daddr);
            let src_offset = ETH_HDR_LEN + offset_of!(iphdr, saddr);

            if start + dest_offset + len > end || start + src_offset +
    len > end{
                return Ok(xdp_action::XDP_ABORTED);
            }
            let dest_to_overwrite = (start + dest_offset) as *mut u32;
            let src_to_overwrite = (start + src_offset) as *mut u32;

            unsafe{
                *dest_to_overwrite = u32::from_be(addr.address1);
                *src_to_overwrite = u32::from_be(addr.address2);
                bpf_redirect_map(&DEV_MAP, &0, XdpAction::PASS);
            }
        },
        // not found in map
```

```
50          None => action = xdp_action::XDP_PASS,
51      }
52  ...
```

Listing 4.20: *Metodo principale con modifica dati del pacchetto e chiamata alla funzione per fare la redirezione*

Capitolo 5

Risultati Sperimentali

Contents

L'obbiettivo di questa tesi era quello di estendere il *Message-oriented Middleware* affinché facesse routing di pacchetti con efficienza elevata, garantendo soprattutto che le richieste con *QoS* elevata, ad esempio di tipo *Strict*, vengano redirette in tempi limitati. In questo capitolo verranno descritti i test effettuati per la misurazione delle prestazioni, in particolare relativamente alla latenza introdotta dalla fase di routing. Verranno inoltre discussi i comportamenti e i risultati nelle diverse modalità di esecuzione di XDP.

5.1 Modalità di esecuzione dei test e risultati

Per il testing della latenza *end-to-end* si è utilizzata un'infrastruttura composta da tre nodi:

- nodo A: si potrebbe ipotizzare come un *Publisher* nell'infrastruttura
- nodo B: per l'esecuzione del *Message-oriented Middleware*
- nodo C: potrebbe essere considerato un *Subscriber*

Il nodo A sarà dunque il generatore di messaggi, il nodo B sarà in grado di ridirigere i pacchetti ricevuti dal nodo A, mentre il nodo C sarà il nodo che riceverà i pacchetti rediretti dal nodo B. Inizialmente l'idea per la misurazione della latenza è stata di calcolare il *round-trip time* di un pacchetto ICMP che eseguisse tutto il percorso (messaggio di *request* dal nodo A attraverso il nodo B per arrivare poi al nodo C e il messaggio di *reply* per il percorso inverso). Per il calcolo del tempo di latenza sono quindi stati paragonati i tempi di invio e di ricezione di una serie di pacchetti

inviati e rediretti, poi messi a confronto con i tempi di un percorso diretto tra nodo A e nodo C (senza quindi passare per il *MoM*). La latenza introdotta sarebbe stata calcolata dividendo per due il *round-trip time* con redirezione mettendolo a confronto con il tempo dell'invio del pacchetto direttamente al nodo C. Il test è stato eseguito inviando un pacchetto ogni 200 ms per un totale di 50.000 pacchetti e calcolandone infine la media. I risultati ottenuti in questa modalità di testing hanno evidenziato i seguenti risultati:

- pacchetto diretto (nodo A verso nodo C):
  rtt min/avg/max/mdev = 0.120/0.225/0.321/0.012 ms
- pacchetto con redirezione (nodo A verso nodo C passando dal nodo B) in modalità di esecuzione native:
  rtt min/avg/max/mdev = 0.194/0.371/0.533/0.040 ms
- pacchetto con redirezione (nodo A verso nodo C passando dal nodo B) in modalità di esecuzione generic:
  rtt min/avg/max/mdev = 0.171/0.363/0.546/0.041 ms

Seppur i test effettuati forniscano un'idea della latenza introdotta dalla redirezione, i tempi ottenuti risultano molto variabili, inoltre, la differenza tra le modalità di esecuzione di XDP non risulta apprezzabile. Il test successivo si basa sulla stessa modalità ma escludendo i tempi di attesa che potrebbero esserci a causa dell'implementazione del comando *ping*. Si è quindi proceduto generando solamente pacchetti UDP ed eseguendo il percorso inverso manualmente, in particolare:

- nodo A: generatore di pacchetti UDP
- nodo B: esegue il programma XDP con la particolarità di redirigere i pacchetti ricevuti dal nodo A verso il nodo C e viceversa (mantenendo le informazioni sulla mappa in modo da avere l'overhead dell'accesso alla mappa in entrambe le direzioni)
- nodo C: esegue un secondo programma XDP scritto appositamente per il test, che ritorna XDP_TX dopo aver modificato i vari indirizzi (dunque rimbalza il pacchetto)

In questa modalità si è ottenuta meno varianza nei risultati, in particolare la media risulta:

- pacchetto diretto (dal nodo A verso il nodo C): 0.172 ms
- pacchetto con redirezione (dal nodo A verso il nodo C passando dal nodo B) in modalità di esecuzione native: 0.340 ms

- pacchetto con redirezione (dal nodo A verso il nodo C passando dal nodo B) in modalità di esecuzione generic: 0.345 ms

Test sicuramente più significativo ma che non evidenziano ancora la differenza nelle due modalità di esecuzione. A tal proposito, si è deciso di generare un numero di pacchetti molto elevato per unità di tempo. In questo modo si è cercato di arrivare ad una saturazione della coda di messaggi in arrivo sul *Message-oriented Middleware* per vedere l'efficienza delle diverse modalità di esecuzione di XDP. Attraverso un programma Rust sono stati mandati 100.000 pacchetti (inizialmente con *sleep*, poi senza) e si è raccolto il dato relativo al numero di pacchetti che sono arrivati a destinazione senza essere scartati . Il test è stato eseguito nelle due diverse modalità di esecuzione di XDP e i dati riportati sono relativi a un ciclo che invia pacchetti senza eseguire una *sleep* tra un invio e l'altro.

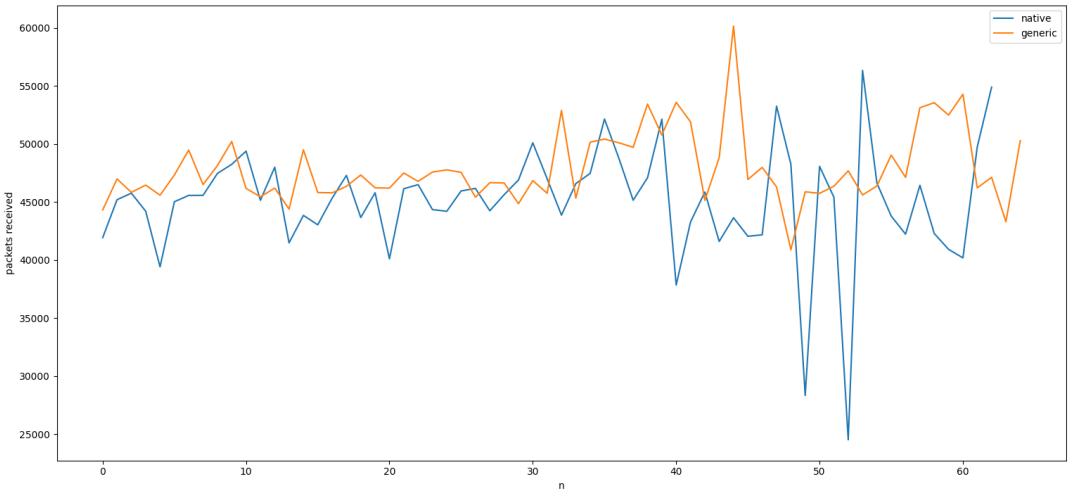

Figura 5.1: *Ricezione di pacchetti su 100.000 pacchetti inviati nelle due modalità di esecuzione di XDP*

Infine, si è misurato il tempo di esecuzione del programma XDP nelle due diverse modalità, dall'arrivo del pacchetto alla redirezione dello stesso. Inizialmente inviando un messaggio per volta e successivamente utilizzando il programma Rust scritto in precedenza.

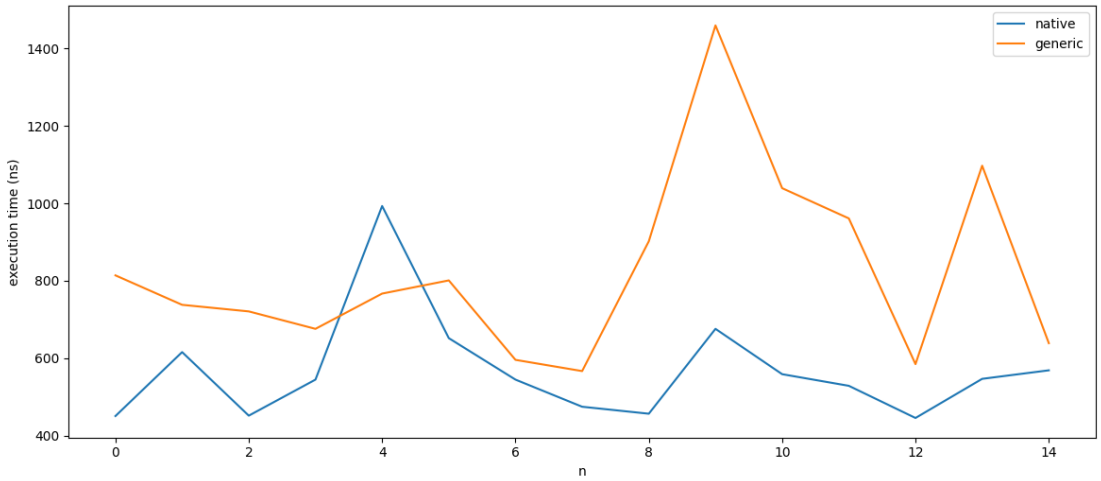

Figura 5.2: *Tempi di esecuzione del programma XDP dall'arrivo del pacchetto alla redirezione dello stesso (un messaggio per volta)*

Nel caso in cui si mandino i pacchetti con una frequenza molto elevata i tempi di esecuzione si riducono drasticamente (40-45 ns per la modalità native e 30-35 ns per la modalità generic) presentando anche una variabilità decisamente inferiore.

5.2 Discussione risultati

Sono state messe a confronto solo due modalità di esecuzione di XDP per la redirezione di pacchetti, questo perché la terza, e più efficiente, richiede una particolare scheda di rete che supporti l'offload su hardware. Quello che si può constatare nei test effettuati, è che la latenza introdotta dal *Message-oriented Middleware* è minima in entrambe le modalità di esecuzione di XDP. Nonostante le piccole differenze per i tempi di latenza ottenute nei tempi di esecuzione per le modalità *native* e *generic*, non sono ritenute sufficienti ad evidenziare una differenza apprezzabile. Tuttavia durante l'esecuzione di tutti i test e osservando i risultati, come ad esempio i grafici della Figura 5.1 e della Figura 5.2, si è potuto constatare che la modalità *native* abbia un comportamento più affidabile rispetto alla modalità *generic*, soprattutto in presenza di invio di messaggi discontinui nel tempo. Si può notare infatti, come in Figura 5.1, la modalità *generic* presenti dei picchi in negativo. Così come per il tempo di esecuzione nella Figura 5.2, che risulta in media superiore al tempo di esecuzione nella modalità *native*. In questo caso i messaggi non sono mandati costantemente, ma distanziati nel tempo. Tra le due modalità il guadagno più significativo si ha sul momento di intercettazione del pacchetto, che avverrebbe prima nel caso in cui il programma risieda direttamente sul driver della scheda di rete. Discorso diverso per la modalità *offload*, dove il pacchetto viene intercettato e processato utilizzando

direttamente l'hardware della scheda di rete e senza nemmeno dover interfacciarsi con la CPU del calcolatore.

Conclusioni e sviluppi futuri

Il lavoro di questa tesi ha permesso lo studio e l'approfondimento della tecnologia eBPF, in particolare del modulo XDP, un processore di pacchetti sicuro, preformante, programmabile e integrato nel kernel. XDP in questo caso è stato usato per l'accelerazione della fase di routing di un Message-oriented Middleware precedentemente progettato per garantire la qualità del servizio (QoS). Il Message-oriented Middleware in questione è parte della sezione Delivery del middleware TEMPOS. È infatti importante che sia garantita l'efficienza durante la fase di routing per le diverse QoS, in particolare per le richieste di qualità del servizio elevate riducendo al minimo la latenza introdotta dalla fase di routing da parte del Message-oriented Middleware.

Un programma XDP esegue all'interno di quello che si chiama *kernel space* e il programma scritto per questa tesi si è rivelato essere molto efficiente nonostante l'accesso ad una struttura dati condivisa con lo *user space* per ogni pacchetto in ingresso. Il programma XDP è infatti in grado di accedere alle informazioni del pacchetto per eseguire una ricerca sulla struttura dati, precedentemente popolata, e determinare poi quali sono gli indirizzi a cui è possibile eseguire il forwarding del pacchetto. I vantaggi ottenuti grazie all'utilizzo di XDP sono:

- **Efficienza**: eseguendo nello spazio kernel e anticipando il pacchetto prima che questo arrivi allo stack di rete, si evitano le copie del pacchetto verso lo spazio utente e altre procedure che risulterebbero in un rallentamento significativo della gestione del pacchetto. Grazie a XDP si è riusciti ad analizzare modificare e redirigere il pacchetto senza dover entrare nello *user space*

- **Sicurezza**: un programma XDP deve essere scritto per essere estremamente sicuro, essendo un codice che esegue all'interno del kernel del sistema operativo. A tal proposito, esiste un componente di eBPF scritto in C in grado di eseguire i controlli prima del caricamento del programma nel kernel

- **Flessibilità**: il pacchetto in ingresso può essere trattato in diversi modi, letto, analizzato, modificato etc., in seguito si può decidere di scartarlo, redirigerlo, passarlo allo stack di rete e così via. Da qui l'aggettivo programmabile, in-

fatti, un programma XDP può adempiere a diversi compiti come *Firewalling*, mitigazione attacchi *DDoS*, load balancing etc.

XDP si presenta in tre diverse modalità di esecuzione: *native*, *generic* e *offloaded*. Per questa tesi si sono testate solamente le prime due in quanto per la terza modalità è necessario avere una scheda di rete che supporti la modalità. La modalità *native* a differenza della modalità *generic* permette il caricamento del programma sul driver della scheda di rete garantendo efficienza maggiore. Per questa tesi entrambe le modalità hanno ottenuto latenze minime con la differenza che la modalità *generic* risultasse meno performante e affidabile al cambiamento di alcune condizioni al contorno. Ad esempio, si è dimostrata avere un degrado delle performance più significativo, rispetto alla modalità *native*, nel caso in cui i messaggi generati non siano costanti nel tempo.

Per la decisione sull'insieme di indirizzi al quale redirigere i vari pacchetti si è utilizzato un file locale di configurazione di tipo TOML. In futuro nel caso in cui si voglia rendere l'infrastruttura più distribuita, eventualmente accessibile da diversi nodi, si potrebbe rendere queste informazioni accessibili tramite l'accesso a un database. Inoltre, si potrebbero fornire schede che supportino l'*offload* su hardware del programma XDP in modo da avere il massimo risultato in termini di efficienza, avendo tempi di latenza inferiori e capacità di carico maggiori. Attualmente la redirezione del pacchetto avviene rimandando lo stesso sulla coda dei pacchetti in uscita della stessa interfaccia. Nel caso in cui in un futuro si volesse aggiungere il supporto alla redirezione attraverso molteplici interfacce di rete, si può utilizzare una delle *helper function* messe a disposizione da eBPF con l'eventuale utilizzo di una struttura dati contenente le interfacce utilizzabili.

Ringraziamenti

Al termine di questa tesi vorrei ringraziare alcune persone che mi hanno sostenuto e supportato per tutto il percorso di studi. Ringrazio il Vice per esserci sempre stato e per il supporto continuo, la mia famiglia senza i quali non sarei mai arrivato fino a questo punto, il professor Bellavista che mi ha accompagnato per il percorso di tesi e in generale per il percorso universitario e il Dott. Garbugli e il dott. Sabbioni, correlatori di tesi, per il supporto durante tutta la stesura di questa tesi.

Bibliografia

[1] Paul Chaignon. *Complexity of the BPF Verifier*. https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html.

[2] Paul Chaignon. *The Cost of BPF Tail Calls*. https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html.

[3] Andrea Garbugli Andrea Sabbioni Antonio Corradi e Paolo Bellavista. «TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things». In: ().

[4] Joanna F. DeFranco e Mohamad Kassab. *What Every Engineer Should Know About the Internet of Things*. 2022.

[5] *eBPF*. https://ebpf.io/what-is-ebpf.

[6] *eBPF maps*. https://www.kernel.org/doc/html/latest/bpf/maps.html.

[7] *eBPF verifier*. https://www.kernel.org/doc/html/latest/bpf/verifier.html.

[8] David Calavera Lorenzo Fontana. *Linux Observability with BPF, Microsoft Azure, and Google Cloud*. 2019.

[9] Red Hat. *What is Serverless?* https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless. 2017.

[10] Netronome Jiong Wang. *Demystify eBPF JIT Compiler*. https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf. 2019.

[11] Theo Lynn, John G. Mooney, Brian Lee e Patricia Takako Endo. *The Cloud-to-Thing Continuum Opportunities and Challenges in Cloud, Fog and and Edge Computing*. 2020.

[12] Microsoft. *What is Pub/Sub Messaging?* https://aws.amazon.com/it/pub-sub-messaging.

[13] Christian Renaud. *The Edge-to-Cloud Continuum*. https://www.delltechnologies.com/asset/en-hk/products/dell-technologies-cloud/industry-market/the-edge-to-cloud-continuum.pdf. 2021.

[14] Maddie Stigler. *Beginning Serverless Computing - Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. 2018.

[15] Giacomo Veneri e Antonio Capasso. *Hands-On Industrial Internet of Things - Create a powerful Industrial IoT infrastructure using Industry 4.0*. 2018.